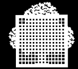
# Bull

## Performance Toolbox Version 2 and 3
## Guide and Reference

AIX

ORDER REFERENCE
86 A2 83EM 00



# Bull

## Performance Toolbox Version 2 and 3 Guide and Reference

AIX

---

Software

September 2004

**BULL CEDOC**
**357 AVENUE PATTON**
**B.P.20845**
**49008 ANGERS CEDEX 01**
**FRANCE**

ORDER REFERENCE
86 A2 83EM 00

The following copyright notice protects this book under the Copyright laws of the United States of America and other countries which prohibit such actions as, but not limited to, copying, distributing, modifying, and making derivative works.

Copyright © Bull S.A. 1992, 2004

Printed in France

Suggestions and criticisms concerning the form, content, and presentation of this book are invited. A form is provided at the end of this book for this purpose.

To order additional copies of this book or other Bull Technical Publications, you are invited to use the Ordering Form also provided at the end of this book.

## Trademarks and Acknowledgements

We acknowledge the right of proprietors of trademarks mentioned in this book.

AIX® is a registered trademark of International Business Machines Corporation, and is being used under licence.

UNIX is a registered trademark in the United States of America and other countries licensed exclusively through the Open Group.

Linux is a registered trademark of Linus Torvalds.

*The information in this document is subject to change without notice. Bull will not be liable for errors contained herein, or for incidental or consequential damages in connection with the use of this material.*

# Contents

**About This Book** . . . ix
Subreleases . . . ix
Content of This Book . . . ix
Highlighting . . . ix
Case-Sensitivity in AIX . . . x
ISO 9000 . . . x
Related Publications . . . x

**Chapter 1. Performance Toolbox for AIX Overview** . . . 1
Why Performance Toolbox for the Operating System? . . . 1
Product Components . . . 3

**Chapter 2. Monitoring Statistics with xmperf** . . . 7
Performance Monitoring . . . 7
Introducing xmperf . . . 7
Monitoring Hierarchy . . . 9
Statistics and Values . . . 10
Instruments . . . 12
Consoles . . . 20
Environments . . . 23
Monitoring Remote Systems with xmperf . . . 23

**Chapter 3. The xmperf User Interface** . . . 29
The xmperf User Interface Overview . . . 29
The xmperf Command Line . . . 29
The xmperf Main Window . . . 31
The Help Menu . . . 33
Console Windows . . . 33
Playback Console Windows . . . 43
Important xmperf Dialogs . . . 46
Tabulating Windows . . . 50

**Chapter 4. Recording and Playback with xmperf** . . . 53
Recording of Statistics . . . 53
Playback of Recordings . . . 55
Using the Playback Console . . . 56
Recording File Inconsistencies . . . 58
Annotations . . . 58

**Chapter 5. The xmperf Command Menu Interface** . . . 61
Command Menus . . . 61
Defining Menus . . . 61
Executables . . . 62
Process Controls . . . 67

**Chapter 6. 3D Monitor** . . . 71
Overview of the 3dmon Program . . . 71
The 3dmon Command Line . . . 74
Customizing the 3dmon Program . . . 76
Recording from 3dmon . . . 80

**Chapter 7. 3D Playback** . . . 83
Overview of the 3dplay Program . . . 83

© Copyright IBM Corp. 1994, 2004

The 3dplay User Interface . . . 84

**Chapter 8. Monitoring Exceptions with exmon** . . . 85
The exmon Main Window . . . 85
The exmon Monitoring Window . . . 85
The exmon Main Window Menu Bar . . . 86
Working with Exception Logs . . . 86
Working with Hosts . . . 87
Command Execution from exmon . . . 89
The exmon Resource File . . . 90

**Chapter 9. Recording Files, Annotation Files, and Recording Support Programs** . . . 93
Recording Files . . . 93
Annotation Files . . . 95
The a2ptx Recording Generator . . . 95
The ptxmerge Merge Program . . . 96
The ptxsplit Split Program . . . 98
The ptxconv Conversion Program . . . 99
Listing Recorded Data with ptxtab . . . 100
The ptxls List Program . . . 102
The ptxrlog Recording Program . . . 103
Listing Recorded HotSet Data with ptxhottab . . . 105
Processing HotSet Recordings with ptx2stat . . . 105

**Chapter 10. Analyzing Performance Recordings with azizo** . . . 107
Initial Processing of Recording Files . . . 107
The azizo Main Window . . . 108
Main Graphs . . . 109
The azizo Command Line . . . 110
The azizo User Interface . . . 110
Using the azizo Metrics Selection Window . . . 114
Working with azizo Main Graphs . . . 117
Common azizo Dialog Boxes . . . 125
Overview of Valid Drag-and-Drop Operations . . . 130

**Chapter 11. Analyzing Performance Trend Recordings with the jazizo Tool** . . . 139
Recording Files . . . 139
Configuration Files . . . 140
Jazizo Tool Menus . . . 140
Legend Panel . . . 145

**Chapter 12. Analyzing WLM with wlmperf** . . . 147
The wlmperf Command . . . 147
Analysis Overview . . . 147
WLM Report Browser . . . 148
Report Properties Panel . . . 148
Report Displays . . . 149
Daemon Recording and Configuration . . . 150
Files . . . 150
Prerequisite . . . 151
Exit Status . . . 151
Related Information . . . 151

**Chapter 13. Monitoring Remote Systems** . . . 153
The System Performance Measurement Interface . . . 153
The xmservd Command Line . . . 155

The xmservd Interface . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 157
The xmquery Network Protocol . . . . . . . . . . . . . . . . . . . . . . . . . 159
Limiting Access to Data Suppliers . . . . . . . . . . . . . . . . . . . . . . . 164
Starting Dynamic Data-Supplier Programs . . . . . . . . . . . . . . . . . . . . 165
Adjusting Socket Buffer Pool . . . . . . . . . . . . . . . . . . . . . . . . . 166

**Chapter 14. Recording Performance Data on Remote and Local Systems** . . . . . . . . . 167
Recording on Remote and Local Systems Overview . . . . . . . . . . . . . . . . . 167
Recording Configuration File . . . . . . . . . . . . . . . . . . . . . . . . . 168
Selecting Metrics for the Recording Configuration File . . . . . . . . . . . . . . 173
The xmscheck Preparser . . . . . . . . . . . . . . . . . . . . . . . . . . . 174
Starting Recording Sessions from the xmtrend Command Line . . . . . . . . . . . . 175
Session Recovery by the xmtrend Agent . . . . . . . . . . . . . . . . . . . . . 176

**Chapter 15. SNMP Multiplex Interface** . . . . . . . . . . . . . . . . . . . . 177
Network Management Principles . . . . . . . . . . . . . . . . . . . . . . . . 177

**Chapter 16. Data Reduction and Alarms with filtd** . . . . . . . . . . . . . . . . 183
filtd Configuration File . . . . . . . . . . . . . . . . . . . . . . . . . . . 183
Data Reduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 184
Defining Alarms . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 187
Using Raw Values and Delta Values . . . . . . . . . . . . . . . . . . . . . . 189

**Chapter 17. Response Time Measurement** . . . . . . . . . . . . . . . . . . . . 191
Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 191
IP Response Time Measurement . . . . . . . . . . . . . . . . . . . . . . . . 192
Application Response Time Measurement (ARM) . . . . . . . . . . . . . . . . . 195

**Chapter 18. System Performance Measurement Interface Programming Guide** . . . . . . . . 201
SPMI Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 201
Understanding the SPMI Data Hierarchy . . . . . . . . . . . . . . . . . . . . 202
Understanding SPMI Data Areas . . . . . . . . . . . . . . . . . . . . . . . . 204
Using the System Performance Measurement Interface API . . . . . . . . . . . . . 216
Example of an SPMI Data User Program . . . . . . . . . . . . . . . . . . . . . 230
Example of an SPMI Data Traversal Program . . . . . . . . . . . . . . . . . . . 234
Example of an SPMI Dynamic Data-Supplier Program . . . . . . . . . . . . . . . 237
SPMI Interface Subroutines . . . . . . . . . . . . . . . . . . . . . . . . . . 241
List of SPMI Error Codes . . . . . . . . . . . . . . . . . . . . . . . . . . . 242

**Chapter 19. Remote Statistics Interface Programming Guide** . . . . . . . . . . . . . 245
Remote Statistics Interface API Overview . . . . . . . . . . . . . . . . . . . . 245
Remote Statistics Interface List of Subroutines . . . . . . . . . . . . . . . . . 246
RSI Interface Concepts and Terms . . . . . . . . . . . . . . . . . . . . . . . 247
A Simple Data-Consumer Program . . . . . . . . . . . . . . . . . . . . . . . 251
Expanding the Data-Consumer Program . . . . . . . . . . . . . . . . . . . . . 254
Inviting Data Suppliers . . . . . . . . . . . . . . . . . . . . . . . . . . . 255
A Full-Screen, Character-based Monitor . . . . . . . . . . . . . . . . . . . . . 257
List of RSi Error Codes . . . . . . . . . . . . . . . . . . . . . . . . . . . . 257

**Chapter 20. Top Monitoring** . . . . . . . . . . . . . . . . . . . . . . . . . 261
Top Monitoring Configuration . . . . . . . . . . . . . . . . . . . . . . . . . 261
Using the jtopas System-Monitoring Tool . . . . . . . . . . . . . . . . . . . . 262

**Appendix A. Installing the Performance Toolbox for AIX** . . . . . . . . . . . . . . 267
Prerequisites . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 267
Installation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 268

Installing Performance Toolbox for AIX on Systems Other Than IBM RS/6000 Hosts . . . . . . . . 268

**Appendix B. Performance Toolbox for AIX Files** . . . . . . . . . . . . . . . . . . . . . 271
Files used by xmperf and Other Data Consumers . . . . . . . . . . . . . . . . . . . . 271
Explaining the xmperf Configuration File . . . . . . . . . . . . . . . . . . . . . . 272
The xmperf Resource File . . . . . . . . . . . . . . . . . . . . . . . . . . . . 277
The azizo Resource File . . . . . . . . . . . . . . . . . . . . . . . . . . . . 282
Simple Help File Format . . . . . . . . . . . . . . . . . . . . . . . . . . . . 286

**Appendix C. Performance Toolbox for AIX Commands** . . . . . . . . . . . . . . . . . 289
3dmon Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 290
3dplay Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 292
a2ptx Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 292
azizo Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 293
chmon Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 293
filtd command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 294
ptxconv Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 294
ptxmerge Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 294
ptxrlog Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 295
ptxsplit Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 296
ptxtab Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 297
xmpeek Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 299
xmperf Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 300
xmscheck Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 302
xmservd Command . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 303

**Appendix D. ARM Subroutines and Replacement Library Implementation** . . . . . . . . . . 305
ARM Subroutines . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 305
ARM Replacement Library Implementation . . . . . . . . . . . . . . . . . . . . . 314

**Appendix E. SPMI Subroutines** . . . . . . . . . . . . . . . . . . . . . . . . . 327
SpmiAddSetHot Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . 327
SpmiCreateHotSet . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 330
SpmiCreateStatSet Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . 331
SpmiDdsAddCx Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . 332
SpmiDdsDelCx Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . 333
SpmiDdsInit Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . 335
SpmiDelSetHot Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . 336
SpmiDelSetStat Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . 338
SpmiExit Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 339
SpmiFirstCx Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . 340
SpmiFirstHot Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . 341
SpmiFirstStat Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . 342
SpmiFirstVals Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . 343
SpmiFreeHotSet Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . 344
SpmiFreeStatSet Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . 345
SpmiGetCx Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 347
SpmiGetHotSet Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . 348
SpmiGetStat Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . 349
SpmiGetStatSet Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . 350
SpmiGetValue Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . 352
SpmiInit Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 353
SpmiInstantiate Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . 355
SpmiNextCx Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 356
SpmiNextHot Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . . 357
SpmiNextHotItem Subroutine . . . . . . . . . . . . . . . . . . . . . . . . . . . 359

SpmiNextStat Subroutine . . . . . 361
SpmiNextVals Subroutine . . . . . 362
Error Codes . . . . . 363
SpmiNextValue Subroutine . . . . . 363
SpmiPathAddSetStat Subroutine . . . . . 365
SpmiPathGetCx Subroutine . . . . . 367
SpmiStatGetPath Subroutine . . . . . 368

**Appendix F. RSi Subroutines** . . . . . 371
RSi Subroutines . . . . . 371

**Appendix G. Notices** . . . . . 409
Trademarks . . . . . 410

**Glossary** . . . . . 413

**Index** . . . . . 415

# About This Book

The Performance Toolbox Guide and Reference provides experienced system administrators with complete detailed information about performance monitoring in a network environment that uses the AIX operating system. Some of the topics included are load monitoring, analysis and control, and capacity planning. This publication is also available on the CD that is shipped with the product.

This book describes Version 3 of the Performance Toolbox for AIX.

## Subreleases

The Performance Toolbox for AIX consists of two components called the Manager and the Agent. The Agent is also referred to as the Performance Aide and represents the component that is installed on every network node in order to enable monitoring by the manager.

Performance Toolbox and the Performance Aide are supported on AIX 5.1 and above. The toolbox versions include different packages:

| PTX Version 3 | | |
|---|---|---|
| **Manager component** | **Type of function** | **Filesets to install** |
| Performance Toolbox | A manager capable of monitoring systems in a distributed environment. | perfmgr.common<br>perfmgr.network<br>perfmgr.analysis |
| Performance Aide | Provides remote agent function for monitoring and recording. | perfagent.server |

The Agent component is available separately from the Performance Toolbox for AIX product.

The Local Performance Analysis and Control Commands fileset (**perfagent.tools**) is now a prerequisite of the Performance Aide for AIX fileset (**perfagent.server**). The Local Performance Analysis and Control Commands ship with the Base Operating System and must be installed before proceeding with the Performance Aide for AIX installation.

## Content of This Book

This edition of this book contains no technical changes. The content is identical to the previous edition.

## Highlighting

The following highlighting conventions are used in this book:

| | |
|---|---|
| **Bold** | Identifies commands, subroutines, keywords, files, structures, directories, and other items whose names are predefined by the system. Also identifies graphical objects such as buttons, labels, and icons that the user selects. |
| *Italics* | Identifies parameters whose actual names or values are to be supplied by the user. |
| `Monospace` | Identifies examples of specific data values, examples of text similar to what you might see displayed, examples of portions of program code similar to what you might write as a programmer, messages from the system, or information you should actually type. |

© Copyright IBM Corp. 1994, 2004

## Case-Sensitivity in AIX

Everything in the AIX operating system is case-sensitive, which means that it distinguishes between uppercase and lowercase letters. For example, you can use the **ls** command to list files. If you type `LS`, the system responds that the command is ″not found.″ Likewise, **FILEA**, **FiLea**, and **filea** are three distinct file names, even if they reside in the same directory. To avoid causing undesirable actions to be performed, always ensure that you use the correct case.

## ISO 9000

ISO 9000 registered quality systems were used in the development and manufacturing of this product.

## Related Publications

The following publication contains information about some performance monitoring and analysis tools:

- *AIX 5L Version 5.3 Performance Management Guide*
- *AIX 5L Version 5.3 Performance Tools Guide and Reference*

# Chapter 1. Performance Toolbox for AIX Overview

This chapter provides an overview to the Performance Toolbox for AIX (PTX) and Performance Aide for AIX (PAIDE) products.

## Why Performance Toolbox for the Operating System?

Anyone faced with the task of keeping a computer system well-tuned and capable of performing as expected recognizes the following areas as essential for success:

**Load Monitoring**
: Resource load must be monitored so performance problems can be detected as they occur or (preferably) predicted well before they do.

**Analysis and Control**
: When a performance problem is encountered, the proper tools must be selected and applied so that the nature of the problem can be understood and corrective action taken.

**Capacity Planning**
: Long term capacity requirements must be analyzed so sufficient resources can be acquired well before they are required.

This book describes the Performance Toolbox for AIX which is designed to help a system administrator do exactly this. The Performance Toolbox for AIX is divided into two components:

- “The Agent” on page 3
- “The Manager” on page 4.

The two components constitute the server (Agent) and the client (Manager) sides of a set of performance management tools, which allow performance monitoring and analysis in a networked environment.

Legacy programs of the Manager component are all X Window System based programs developed with the OSF/Motif Toolkit. One program, **xmperf**, is at the same time a graphical monitor, a tool for analyzing system load, and an umbrella for other tools, performance related or not. Another monitoring program is **3dmon**, which allows the monitoring of a large number of statistics in a single window. The program **exmon** is designed to work with the **filtd** daemon. It monitors alarms generated by **filtd**.

There are two new Java-based programs in Version 3. The first, **jazizo**, is a post-processing application for analyzing trend recordings made by the **xmtrend** daemon. The second, **jtopas**, is a client for viewing top resource usage on local and remote systems. This application can show same time and recorded data.

The main program in the Agent component contains the three daemons. The **xmservd** daemon acts as a networked supplier of performance statistics and optionally as a supplier of performance statistics to Simple Network Management Protocol (SNMP) managers. The **xmtrend** daemon creates long term and optimized recordings for analysis by the **jazizo** trend and **jtopas** top resource clients. The **filtd** daemon supports the filtering function on the existing statistics pool, allows users to create customized metrics, and generates alarms for the **exmon** exception monitor.

Both components include application programming interfaces for application controlled access and supply of performance statistics.

## Monitoring Features

The client/server environment allows any program, whether part of Performance Toolbox for AIX or custom developed applications, to monitor the local host as well as multiple remote hosts. This ability is fully explored in the Manager component program **xmperf** whose “monitors” are graphical windows, referred to as *consoles*, which can be customized on the fly or kept as pre-configured consoles that can be invoked

© Copyright IBM Corp. 1994, 2004

with a few mouse clicks. Consoles can be generic so the actual resource to monitor, whether a remote host, a disk drive, or a LAN interface, is chosen when the console is opened. Consoles can be told to do a recording of the data they monitor to disk files and such recordings can be played back with **xmperf** and analyzed with the **jazizo** program.

One of the things that makes **xmperf** unique is that it is not hardcoded to monitor a fixed set of resources. It is dynamic in the sense that a system administrator can customize it to focus on exactly the resources that are critical for each host that must be monitored.

An implementation of the Application Response Management (ARM) specification allows applications to be instrumented so that the activity and response times of applications can be monitored. In addition, the raw response time from any host running the PTX agent to any IP capable host in the network can be monitored.

Another feature is the Agent filter called **filtd**, which allows you to easily combine existing "raw" statistics into new statistics that make more sense in your environment. This can be done by entering simple expressions in a configuration file and requires no programming.

## Analysis and Control

By providing an umbrella for tools that can be used to analyze performance data and control system resources, the Manager program **xmperf** assists the system administrator in keeping track of available tools and in applying them in appropriate ways. This is done through a customizable menu interface. Tools can be added to menus, either with fixed sets of command line arguments to match specific situations or such that the system administrator has an easy way to remember and enter command line arguments in a dialog window. The menus of **xmperf** are preconfigured to include most of the performance tools shipped as part of the tools option of the Agent component.

Properly customized, **xmperf** becomes an indispensable repository for tools to analyze and control an operating system. In addition, the ability to record load scenarios and play them back in graphical windows at any desired speed gives new and improved ways of analyzing a performance problem.

Outstanding features for analyzing a recording of performance data are provided by the **jazizo** program and its support programs. Recordings can be produced from the monitoring programs **xmperf** and **3dmon** during monitoring, or can be created by the **xmservd** daemon. This makes constant recording possible so that you can analyze performance problems after they have occurred. The **3dplay** program is provided to play back recordings created by **3dmon** in the same style in which the data was originally displayed.

Finally, using the Agent component filter **filtd**, you can define conditions that, when met, could trigger any action you deem appropriate, including alerting yourself and/or initiating corrective action without human intervention. This facility is entirely configurable so that alarms and actions can be customized to your installation.

## Capacity Planning

If your system is capable of simulating a future load scenario, **xmperf** can be used to visualize the resulting performance of your system. By simulating the load scenario on systems with more resources, such as more memory or disks, the result of increasing the resources can be demonstrated.

## Networked Operation

The **xmservd** data-supplier daemon can provide a stream of data to consumers of performance statistics. Frequency and contents of each packet of performance data are determined by the consumer program. Any consumer program can access performance data from the local host and one or more remote hosts; any data-supplier daemon can supply data to multiple hosts.

In addition to its ability to monitor across a network, PTX also allows for monitoring the response time to and from nodes in the network itself.

## Application Programming Interfaces

Each component comes with its own application programming interface (API). In addition, an API is available for instrumentation of application programs:

**Agent API**
Called the System Performance Measurement Interface (SPMI) (Chapter 18, "System Performance Measurement Interface Programming Guide," on page 201) API. It allows an application program to register custom performance statistics about its own performance or that of some other system component. When registered, the custom statistics become available to any consumer of statistics, local or remote. Programs that supply custom statistics are called dynamic data-supplier programs.

The Agent API also permits applications to access statistics on the local system without using the network interface. Such applications are called local data-consumer programs.

**Manager API**
Called the Remote Statistics Interface (RSi) (Chapter 19, "Remote Statistics Interface Programming Guide," on page 245) *API*. Allows an application program to access statistics from remote nodes (or the local host) through a network interface.

**Application Monitoring API**
Called the Application Response Management (ARM), this API permits application program to be instrumented in such a way that the application activity and response time can be monitored from any of the PTX manager programs.

## SNMP Interface

By entering a single keyword in a configuration file, the data-supplier daemon can be told to export all its statistics to a local **snmpd** SNMP agent. Users of an SNMP manager such as NetView see the exported statistical data as an extension of the set of data already available from **snmpd**.

**Note:** The SNMP multiplex interface is only available on IBM pSeries Agents.

## On Statistics, Metrics, and Values

Throughout this book, the terms *statistic* and *metric* are used to describe a probe in or instrumentation of a component of the operating system or, in some cases, application programs. The probe is usually defined and updated by the system, regardless of the presence of Performance Toolbox for AIX. In the current version, a probe is always represented by a data field that is either incremented each time a certain event occurs (a counter) or represents a quantity, such as the amount of free disk or memory.

In this book, the terms statistic and metric are synonymous. Usually metric is used in connection with the **jazizo** program and related programs. The term statistic is the preferred term in describing the APIs and other programs.

Finally, the term *value* is used to refer to a statistic (metric) when included in a monitoring "device" in the programs **xmperf** and **3dmon**.

# Product Components

## The Agent

The Agent component of the Performance Toolbox for AIX is identical to the Performance Aide for AIX licensed product server option. It has the following main components:

**xmservd**
The data-supplier daemon, which permits a system where this daemon runs to supply performance statistics to data-consumer programs on the local or remote hosts. This daemon also provides the interface to SNMP.

**Note:** The interface to SNMP is available only on pSeries Agents.

**xmtrend**
Long term recording daemon. Provides large metric set trend recordings for post-processing by **jazizo** and **jtopas**.

**xmscheck**
A program that lets you pre-check the **xmservd** recording configuration file. This program is useful when you want to start and stop **xmservd** recording at predetermined times.

**filtd**
A daemon that can be used to do data reduction of existing statistics and to define alarm conditions and triggering of alarms.

**xmpeek**
A program that allows you to display the status of **xmservd** on the local or a remote host and to list all available statistics from the daemon.

**iphosts**
A program to initiate monitoring of Internet Protocol performance by specifying which hosts to monitor. Accepts a list of hosts from the command line or from a file.

**armtoleg**
A program that can convert a pre-existing Application Response Management (ARM) library into an ARM library that can be accessed concurrently with the ARM library shipped with PTX. Only required and available on operating systems.

**SpmiArmd**
A daemon that collects Application Response Management (ARM) data and interfaces to the Spmi library code to allow monitoring of ARM metrics from any PTX manager program.

**SpmiResp**
A daemon that polls for IP response times for selected hosts and interfaces to the Spmi library code to allow monitoring of IP response time metrics from any PTX manager program.

**Application Response Management API and Libraries**
A header file and two libraries support the PTX implementation of ARM. The implementation allows for coexistence and simultaneous use of the PTX ARM library and one previously installed ARM library.

**System Performance Measurement Interface API and Library**
Header files and a library to allow you to develop your own data-supplier and local data-consumer programs.

**Sample Programs**
Sample dynamic data-supplier and data-consumer programs that illustrate the use of the API.

**PTX Agents for selected non-IBM platforms**
Compressed tar files provide full agent support on multiple HP-UX and Solaris Versions on various HP and Sun systems.

## The Manager

The Manager component of the Performance Toolbox for AIX has the following main components:

**xmperf**
The main user interface program providing graphical display of local and remote performance information and a menu interface to commands of your choice.

**3dmon**
A program that can monitor up to 576 statistics simultaneously and display the statistics in a 3-dimensional graph.

**3dplay**
A program to play 3dmon recordings back in a 3dmon-like view.

**chmon**
Supplied as an executable as well as in source form, this program allows monitoring of vital statistics from a character terminal.

**exmon**
The program that allows monitoring of alarms generated by the **filtd** daemon running on remote hosts.

**azizo**
Legacy recording tool replaced by **jazizo** in PTX Version 3. A program that allows you to

analyze any recording of performance data. It lets you zoom-in on sections of the recording and provides graphical as well as tabular views of the entire recording or zoomed-in parts of it.

**jazizo**
Program allows you to analyze long term **xmtrend** recordings of performance data. Reports can be generated displaying system activity over hours, days, weeks or months. Tabular views of the entire recording or subsets are provided. This application is available only in PTX Version 3.

**jtopas**
Provides tabular top resources views of a selected system. Near real-time and recorded data may be viewed. Reports can be generated over various time periods. Utilizes data from **xmtrend** recordings.

**ptxtab**
A program that can format statset date from recording files for printed output.

**ptxmerge**
This program allows you to merge up to 10 recording files into one. For example, you could merge **xmservd** recordings from the client and server sides of an application into one file to better correlate the performance impact of the application on the two sides.

**ptxsplit**
In cases where recording files are too large to analyze as one file, this program allows you to split the file into multiple smaller files for better overview and faster analysis.

**ptxrlog**
A program to create recordings in ASCII or binary format.

**ptxls**
A program to list the control information of a recording file, including a list of the statistics defined in the file.

**a2ptx**
The **a2ptx** program can generate recordings from ASCII files in a format as produced by the **ptxtab** or **ptxrlog** programs or the Performance Toolbox for AIX SpmiLogger sample program. The generated recording can then be played back by **xmperf** or analyzed with **jazizo**.

**ptxconv**
The format of recordings has changed between Versions of the Performance Toolbox for AIX. As a convenience to users of multiple versions of the Performance Toolbox for AIX, this program converts recording files between the formats of the different versions.

**ptx2stat**
Converts hotset data collected in a recording file to a format that resembles the recording format for statsets. Permits postprocessing of hotset data with the programs that allow playback and manipulation of recordings.

**ptxhottab**
A program that can format and print hotset information collected in recording files.

**wlmperf**
Program for analyzing Workload Management (WLM) activity from **xmtrend** recordings. Provides reports on class activity across hours, days or weeks in a variety of formats. This application is available only in PTX Version 3.

**Remote Statistics Interface API**
Header file to allow you to develop your own data-consumer programs.

**Sample Programs**
Sample data-consumer programs that illustrate the use of the API.

# Chapter 2. Monitoring Statistics with xmperf

This chapter provides information about monitoring statistics with the **xmperf** program.

It is beneficial to have **xmperf** working while you read this chapter. To do this, follow the instructions in Appendix A, “Installing the Performance Toolbox for AIX,” on page 267 to install the program and supporting files. When you have successfully installed the files, follow these steps:

1. To start the X Window System, issue the command **xinit**.
2. From the aixterm window, issue the command **xmperf&** to start the program and make it run in the background.
3. Use the pull-down menu marked Monitor to see the list of available monitoring devices. Click on a few to see different ones.
4. Adjust the windows opened by **xmperf** so that they don’t hide each other or the aixterm window.

## Performance Monitoring

Monitoring the performance of computer systems is one of the most important tasks of a system administrator. Performance monitoring is important for many reasons, including:

- The need to identify and possibly improve performance-heavy applications.
- The need to identify scarce system resources and take steps to provide more of those resources.
- The need for load predictions as input to capacity planning for the future.

The **xmperf** performance monitor is a powerful and comprehensive graphical monitoring system. It is designed to monitor the performance on the system where it itself is executing and at the same time allow monitoring of remote systems via a network. Every host to be monitored by **xmperf**, including the system where **xmperf** runs, must have the Agent component installed and properly configured.

**Note:** Remote systems can only be monitored if the Performance Toolbox Network feature is installed. If the Performance Toolbox Local feature is installed then only the local host can be monitored. Discussions about remote system monitoring pertain only to the Performance Toolbox Network feature.

In addition to monitoring performance in semi-real time, **xmperf** also is designed to allow recording of performance data and to play recorded performance data back in graphical windows.

## Introducing xmperf

The **xmperf** program is the most comprehensive and largest program in the Manager component of PTX. It is an X Window System-based program developed with the OSF/Motif Toolkit. The **xmperf** program allows you to define monitoring environments to supervise the performance of the local system and remote systems. Each monitoring environment consists of a number of consoles. Consoles show up as graphical windows on the display. Consoles, in turn, contain one or more instruments and each instrument can show one or more values that are monitored.

Each **xmperf** environment is kept in a separate configuration file. The configuration file defaults to the file name **xmperf.cf** in your home directory but a different file can be specified when **xmperf** is started. See Appendix B, “Performance Toolbox for AIX Files,” on page 271 for alternative locations of this file.

Consoles are named entities allowing you to activate one or more consoles by selecting from a menu. New consoles can be defined and existing ones can be changed interactively. Consoles allow you to define and group collections of monitoring instruments in whichever configuration is convenient. Instruments are the actual monitoring devices enclosed within consoles as rectangular graphical subwindows.

© Copyright IBM Corp. 1994, 2004

There are two kinds of instruments:

**Recording Instruments**
The term *recording* describes the ability to show the statistics for a system resource over a period of time. Recording does not mean that the instruments record events to a disk file; all instrument types can do that. Recording instruments have a time scale with the current time displayed on the right. The values plotted are moved to the left when new readings are received.

**State Instruments**
State instruments show the latest statistics for a system resource, optionally as a weighted average. They do not show the statistics over time but collect this data in case you want to change the instrument to a recording instrument.

All instruments are defined with a primary graph style. The following graph styles are currently available:

**Recording graphs:**

- Line graph
- Area graph
- Skyline graph
- Bar graph

**State graphs:**

- State bar
- State light
- Pie chart
- Speedometer

All instruments can monitor up to 24 different statistics at a time. Each statistic is represented by a value. Values can be selected from menus of available statistics. Depending on your system and the availability of tools, statistics could be local or remote.

To illustrate these concepts, think of a hypothetical example (illustrated in Figure 1 on page 9) of a console defined to contain three instruments as follows:

State instrument, shaped as a pie chart, showing three values as a percentage of total memory in the local system:

- Free memory
- Memory used for computational segments
- Memory used for non-computational segments.

Recording instrument, plotted as a state bar graph, showing four values:

- Percent of CPU time spent in kernel mode
- Percent of CPU time spent in user mode
- Percent of CPU time spent waiting for disk input/output
- Percent of CPU time spent in idle state.

Recording instrument, plotted as a line graph, showing:

- Number of page-ins per second
- Number of page-outs per second.

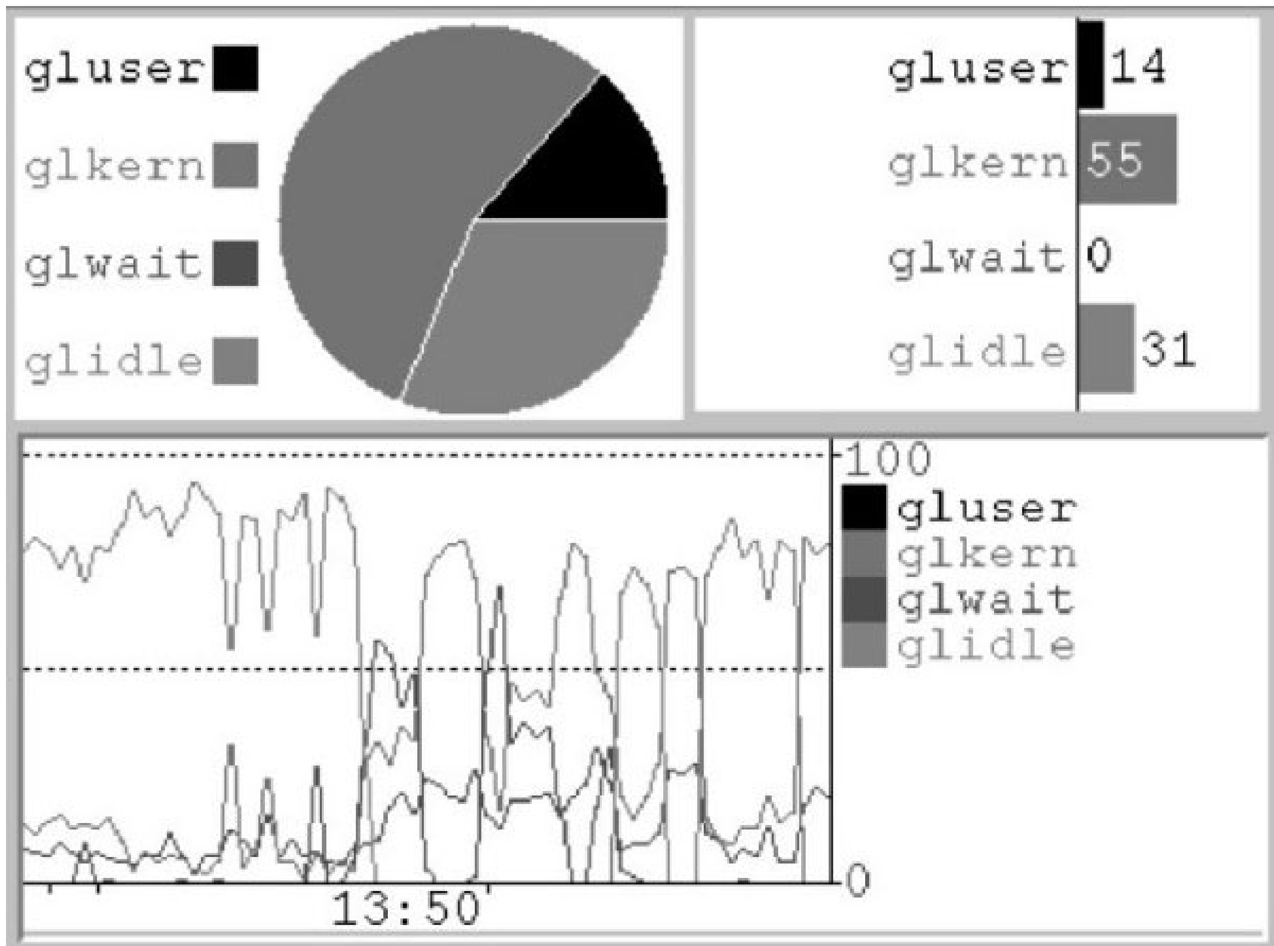



*Figure 1. Sample xmperf Console. This console shows a mixture of chart styles, including a pie chart (upper left), a bar chart (upper right), and a scrolling histograph (bottom). The histograph's horizontal access represents time while the vertical axis is set to a user-specified range from 0 to 100.*

In addition to the monitoring features, **xmperf** also provides an enhanced interface to system commands. Configure the interface by editing the **xmperf** configuration file. Commands can be grouped under either of three main menu items:

- Analysis
- Controls
- Utilities

As another convenience, **xmperf** also can be used to display a list of active processes in the system. The list can be sorted after a number of criteria. Associated with the process list is yet another user-configurable command menu. Commands defined here take a list of process IDs as argument.

## Monitoring Hierarchy

Whenever you start **xmperf**, you must supply a configuration file to define the environment in which you want to do the monitoring. The file can be an empty (zero-length) file, in which case you must define the environment from scratch. If no file is specified, **xmperf** defaults to the file **xmperf.cf** in your home directory. If the file does not exist in your home directory, it is searched for as described in Appendix B, "Performance Toolbox for AIX Files," on page 271. Usually, the configuration file defines one or more consoles, each typically used to monitor a related set of performance data. For example, one console

might be called Network and provide one graph for each network interface in the system. Another might be called Disks and contain graphs to show the activity of each physical disk and the types of input/output requests.

The configuration file, thus, defines the environment and contains one or more definitions of consoles. Together they constitute the top two levels in the hierarchy. The third level is the subdivision of consoles into instruments. Using the previous Network example, this console might have an instrument defined for each of your system's network interfaces. Thus, whenever you want to check the network load, call a monitoring console by its name, in this case Network, and monitoring starts immediately.

Returning to the example of network monitoring, obviously, there's more than one thing to keep an eye on for each of the network interfaces in a system. You might want to monitor the number of retransmissions and probably want to know how much network load comes from input and how much from output. Those are just a couple of the many types of data that might interest you. In **xmperf** terms, each data type is referred to as a *value* and represents a flow of data from one particular type of statistics collected by the system. If each of these data values had to be monitored in separate instruments, it would be difficult to correlate them and you'd end up with an unmanageable number of instruments for even simple monitoring tasks. Therefore, an instrument can be made to monitor more than one value so that each instrument monitors a set of data values. Values comprise the fourth level in the monitoring hierarchy.

## Statistics and Values

A given system, at any point in time, has a number of system resources that do not change between boots. Such resources include, but are not limited to, the following:

- Processing units
- Real memory
- Non-removable disk drives
- Network interfaces.

Your system collects a large amount of performance-related statistics about such resources. This data can be accessed from application programs through the APIs provided by PTX. One such application program is **xmperf**. It provides a user interface that allows you to select the data to monitor from lists of the data available. By selecting multiple data values, you can build instruments where multiple statistics can be plotted on a common time scale for easy correlation.

## Data Value Properties

When you select a value to monitor, you get a set of default properties for that data value. Each property can be changed to reflect special needs. The properties associated with a data value and their defaults are as follows:

**Style**
A secondary graph style, which is ignored for state graphs. By specifying a secondary graph style different from the primary style of an instrument, interesting effects can be created. For example, if the instrument's primary style is a bar graph, then you can choose a line graph style for one or more values to plot related values so they overlay the bar graph.

Default = Same as primary style for instrument.

**Color**
The color used to represent the value when plotted. If the value is displayed as a state light, the color is used to paint the lights when a data value is lower than a descending threshold or higher than an ascending threshold.

Default = As specified in resource file (see "The xmperf Resource File" on page 277) or generated from colors in the color map and contrasting from neighbor colors (for colors not defined in resource file).

**Tile**
A pattern (pixmap) used to tile the value as it is drawn in an instrument. Tiles are ignored

for line drawing, for text, and for the state light type instruments. When tiling is used, it is always done by mixing the color of the value and the background color of the instrument in one out of eleven available patterns, numbered from 1 to 11.

Default = foreground (tile 1 = 100% foreground color).

**Scale, low**
The lowest value plotted. This property only has a meaning for recording graphs and for the state bar graph.

When a low scale value is given for a recording graph or a state bar graph, then the scale of the graph goes from the low scale value to the high scale value. For example, if the low scale is 50 and high scale is 100, then the lowest value you will ever see plotted in the instrument is 51. A value of 75 would extend half-way into the plotting area.

Default = From system tables, usually zero.

**Scale, high**
The value that determines the scale of the graphs. If values are encountered that exceed the high scale limit, the graphs are cut off to fit the plotting area. This property has no meaning for the state light graph type.

Default = From system tables.

**Threshold**
This property is used only for the state light graph type. It defines the value at which the "light is turned on." Whether the light is on, when the value is above or below the threshold is determined by the threshold type.

Default = zero.

**Threshold Type**
This property is used only for the state light graph type. It must be descending or ascending. If descending, the light is turned on when the last value received is equal to or below the threshold. If ascending, the light is turned on when the last value received is equal to or above the threshold.

Default = Ascending.

**Label**
This property can be used to specify a user-defined text that is used to label the value in the instrument.

Default = Null (path name of value is used, see "Path Names").

## Path Names

Many system resources exist in multiple copies. For example, a system can have three disks, and two of those can be used for paging space. A system can also have multiple network interfaces, and several other resources can be duplicated. Generally, one set of statistics is collected for each resource copy.

Because of this duplication of statistics, the selection of values to plot in an instrument is done through a multi-level selection process. Because this process is available, it is also used to group statistics even when they are not duplicated. The unique name used to identify a value is composed of one-level names separated by slashes, much like a fully qualified UNIX file name. The fully qualified name of a value is called the path name of the value. To identify the percentage of time a particular disk on the host system with the hostname **birte** is busy, the path name might be:

```
hosts/birte/Disk/hdisk02/busy
```

For space reasons, it is seldom possible to display all of the path name in instruments. For example, given that the number of characters used to display value names is 12, only the last 12 characters of the above name would be displayed, yielding:

```
hdisk02/busy
```

The default length of a value name is 12, but you can specify up to 32 characters using the command line argument **-w**. In addition, the **-a** command line argument allows you to request *adjustment* of the text

length to what is necessary to display the value names. When **-a** is used, the text length can be less than what is specified by **-w** (or the default, whichever applies) but never longer. The X Window System resources **LegendAdjust** and **LegendWidth** (see “Execution Control Resources” on page 280) can be used in place of the command line arguments. See “The xmperf Command Line” on page 29 for a description of command line options and “The xmperf Resource File” on page 277 for a description of supported resources.

In many cases, an instrument or a console is used to display statistics that have some of the value path name in common. When this happens, **xmperf** automatically removes the common part of the name from the displayed name and shows it in an appropriate place, dependent on the type of instruments used. This is explained in “Value Name Display” on page 17 and “The Console Title Bar” on page 22.

### User-defined Labels

Regardless of your efforts to carefully name each level in the hierarchy of statistics, your results might include some that are not informative. In such cases, you might want to specify your own name for a value. Do this from the dialog box used to add or change a value as described in “Changing the Properties of a Value” on page 48.

## Instruments

An instrument occupies a rectangular area within the window that represents a console. Each instrument may plot up to 24 values simultaneously. The instrument defines a set of statistics and is fed by network packets that contain a reading of all the values in the set, taken at the same time. All values in an instrument must be supplied from the same host system.

The instrument shows the incoming observations of the values as they are received depending on the type of statistic selected. Statistics can be of types:

**SiCounter**
: Value is incremented continuously. Instruments show the delta (change) in the value between observations, divided by the elapsed time, representing a rate per second.

**SiQuantity**
: Value represents a level, such as memory used or available disk space. The actual observation value is shown by instruments.

Instruments defined in **xmperf** correspond to statsets in the **xmservd** daemons of the systems the instruments are monitoring. The section on “Statsets” on page 153 gives information about statsets and their relationship to instruments.

## Configuring Instruments

Instruments can be configured through a menu-based interface as described in “The Modify Instrument Submenu” on page 36. In addition to selecting from 1 to 24 values to be monitored by the instrument, the following properties are established for the instrument as a whole:

**Style**
: The primary graph style of the instrument. If the graph is a recording graph, not all values plotted by the graph need to use this graph style. In the case of state graphs, all values are forced to use the primary style of the instrument.

  Default = `Line graph`.

**Foreground**
: The foreground color of the instrument. Most noticeably used to display time stamps and lines to define the graph limits.

  Default = White.

**Background**
: The background color of the instrument.

  Default = `Black`.

**Tile** A pattern (pixmap) used to tile the background of the instrument. Tiles are ignored for state light type instruments. When tiling is used, it is always done by mixing the foreground color and the background color of the instrument in one out of eleven available patterns.

Default = `Tile 2` (100% background color, that is, no tiling).

**Interval**
The time interval between observations. Minimum is 0.2 second, maximum is 30 minutes. Even though the sampling interval can be requested as any value in the above range, it may be changed by the **xmservd** daemon on the remote system that supplies the statistics. For example, if you request a sampling interval of 0.2 second but the remote host's daemon is configured to send data no faster than every 500 milliseconds, then the remote host determines the speed. As explained in "Rounding of Sampling Interval" on page 156, **xmservd** rounds sampling intervals so that the previous example would result in an effective sampling interval of 500 milliseconds.

Default = `5` seconds.

**History**
The number of observations to be maintained by the instrument. For example, if the interval between observations is 5 seconds and you have specified that the history is 1,000 readings, then the time period covered by the graph is 1,000 x 5 seconds or approximately 83 minutes.

The history property is defined for recording graphs only. If the current size of the instrument is too small to show the entire time period defined by this property, scroll the instrument to view older values. State graphs that show only the latest reading, so their history property is undefined. Because you can change the primary style of an instrument at any time, the actual readings of data values are kept according to the history property. Thus, data is not lost if you change the primary style from a state graph to a recording graph.

The minimum number of observations is 50 and the maximum number you can specify is 5,000.

Default = `500` readings.

**Stacking**
The concept of stacking allows you to have data values plotted on top of each other. Stacking works only for values that use the primary style. To illustrate, think of a bar graph where the kernel-CPU and user-CPU time are plotted as stacked. If at one point in time the kernel-CPU is 15% and the user-CPU is 40%, then the corresponding bar goes from 0-15% in the color of kernel-CPU, and from 16-55% in the color used to draw user-CPU.

If you want to overlay this graph with the number of page-in requests, you could do so by letting this value use the skyline graph style. It is important to know that values are plotted in the sequence they are defined. Thus, if you wanted to switch the CPU measurements given previously, define user-CPU before you define kernel-CPU. Lastly, define values to overlay graphs in a different style to avoid obscuring these values by the primary style graphs.

Default = `No stacking`.

**Shifting**
This property is meaningful for recording graphs only. It determines the number of pixels the graph should move as each reading of values is received. The size of this property has a dramatic influence on the amount of memory used to display the graph because the size of the pixmap (image) of the graph is proportional with the product:

```
history x shifting x graph height
```

If the shifting is set to one pixel, a line graph looks the same as a skyline graph, and an area graph looks the same as a bar graph. Maximum shifting is 20 pixels, minimum is (spacing + 1) pixel.

Default = `4` pixels.

**Spacing**

A property used only for bar graphs and state bar graphs. It defines the number of pixels separating the bar of one reading from the bar of the next. For a bar graph, the width of a bar is always (shifting — spacing) pixels. The property must always be from zero to one less than the number of pixels to shift.

Default = 2 pixels.

In addition to the previously mentioned properties that can be modified through a menu interface, four properties determine the relative position of an instrument within a console. They describe, as a percentage of the console's width and height, where the top, bottom, left and right sides of the instrument are located. In this way, the size of an instrument is defined as a percentage of the size of the monitor window.

The relative position of the instrument can be modified by moving and resizing it as described in "Moving Instruments in a Console" on page 22 and "Resizing Instruments in a Console" on page 21.

### Use of Colors for State Lights

For the state light graph type, foreground and background colors are used in a special way. To understand this, consider that state lights are shown as text labels "stuck" onto a background window area like you would stick paper notes to a bulletin board. The background window area is painted with the foreground color of the instrument rather than with the background color. The color of the background window area never changes.

Each state light may be in one of two states: lit (on) or dark (off). When the light is off, the value is shown with the label background in the instrument's background color and the text in the instrument's foreground color. If the instrument's foreground and background colors are the same, you see only an instrument painted with this color; no text or label outline is visible. If the two instrument colors are different, the labels are seen against the instrument background and label texts are visible.

When the light is on, the instrument's background color is used to paint the text while the value color is used to paint the label background. This special use of colors for state lights allows for the definition of alarms that are invisible when not triggered or alarms that are always visible.

## Skeleton Instruments

Some statistics change over time. The most prominent example of statistics that change is the set of processes running on a system. Because process numbers are assigned by the operating system as new processes are started, you can never know what process number an execution of a program will be assigned. This makes it difficult to define consoles and instruments in the configuration file.

To help you cope with this situation, a special form of consoles can be used to define skeleton instruments. Skeleton instruments are defined as having a "wildcard" in place of one of the hierarchical levels in the path that defines a value. For example, you could specify that a skeleton instrument has the following two values defined as follows:

```
Proc/*/kern
Proc/*/user
```

The wildcard is represented by the asterisk. It appears in the place where a fully qualified path name would have a process ID. Whenever you try to start a console with such a wildcard, you are presented with a list of processes. From this list, you can select one or more instances. To select more than one instance, move the mouse pointer to the first instance you want, then press the left mouse button and move the mouse while holding the button down. When all instances you want are selected, release the mouse button. If you want to select instances that are not adjacent in the list, press and hold the Ctrl key on the keyboard while you make your selection. When all instances are selected, release the Ctrl key.

Skeleton consoles cannot be defined through the menu interface. They must be defined by entering the skeleton console definitions in the **xmperf** configuration file. This is described in “Defining Skeleton Consoles” on page 276.

Each process selected is used to generate a fully qualified path name. If the wildcard represents a context other than the process context, such as disks, remote hosts, or LAN interfaces, the selection list will represent the instances of that other context. In either case, the selection you make is used to generate a list of fully qualified path names. Each path name is then used to define a value to be plotted or to define a new instrument in the console. Whether you get one or the other, depends on the type of skeleton you defined. There are two types of skeleton consoles:


- “Skeleton of Type “All”” on page 16
- “Skeleton of Type “Each”” on page 16

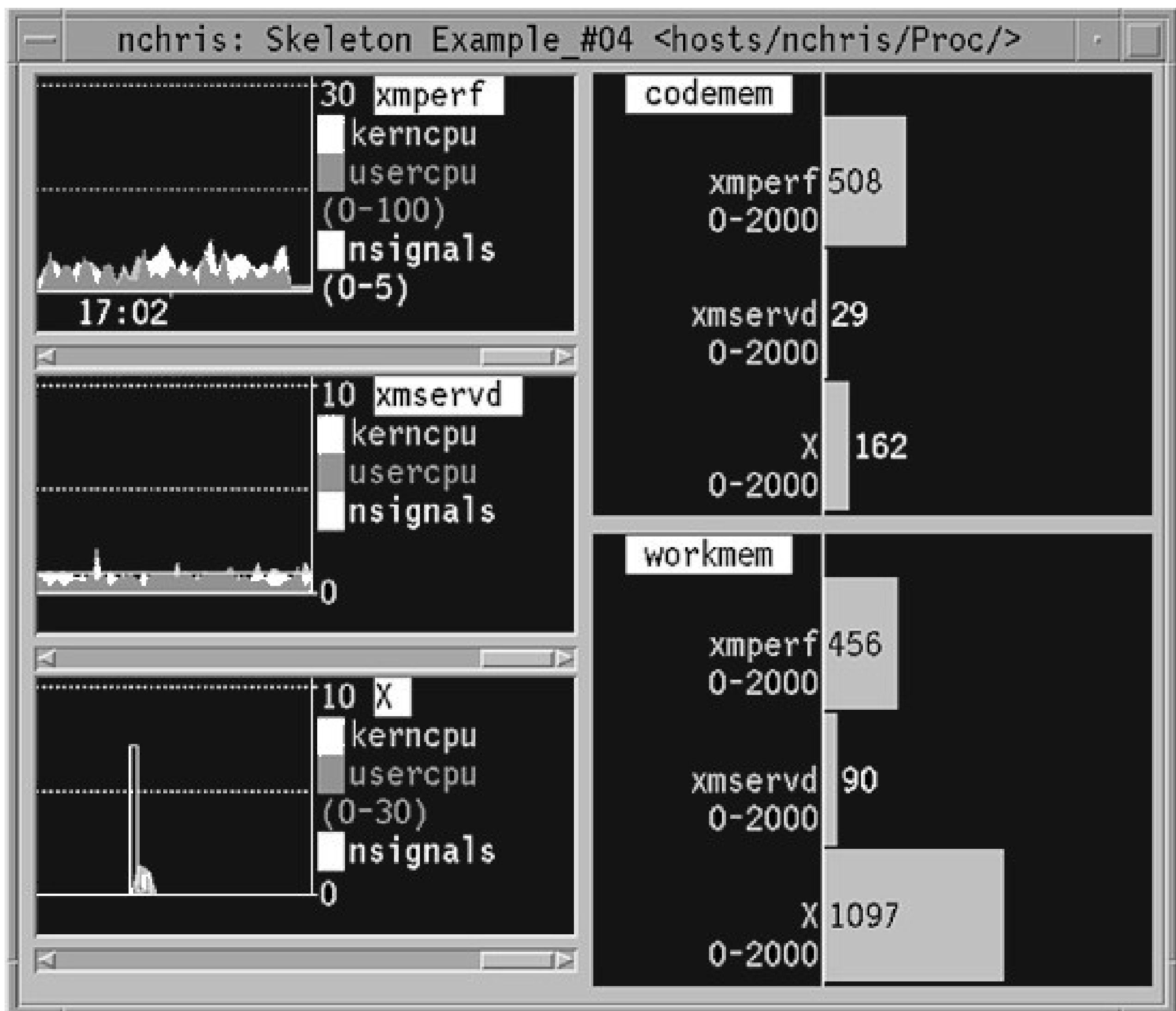




*Figure 2. Instantiated Skeleton Console. This console shows five instruments (appearing as separate windows within the larger window). Three processes (kerncpu, usercpu, and the nsignals) were selected to instantiate the skeleton console; these are shown on the left side using line graphs. On the right side, are two instruments showing three processes (xmperf, wmservd, and X) that instantiate the skeleton console. Horizontal bar graphs are used to display the three processes on the right.*

## Skeleton of Type “All”

The skeleton type named “All” includes all value instances you select into the instrument. A skeleton instrument creates exactly one instance of an instrument and this single instrument contains values for all selected value instances. This is shown in the right side of the instantiated skeleton console shown in the preceding figure, Instantiated Skeleton Console. Two type “All” instruments are defined in the right side of the console, and three processes were selected to instantiate the skeleton console.

Skeleton instruments of type “All” are defined with only one value because the instantiated instrument contains one value for each of the selections you make from the instance list.

Consoles can be defined with both skeleton instrument types but any non-skeleton instrument in the same console is ignored. The relative placement of the defined instruments is kept unchanged. When many value instances are selected, it can result in crowded instruments; but you can resolve this by resizing the console. When all the skeleton instruments in a console are type “All” skeletons, **xmperf** fails to automatically resize the console.

The type of instrument best suited for the type “All” skeleton instruments is the state bar, but other graph types may be useful if you allow colors to be assigned to the values automatically. To do the latter, specify the color as **default** when you define the skeleton instrument.

## Skeleton of Type “Each”

This skeleton type is so named because each value instance that you select creates one instance of the instrument. When you select five value instances, each of the type “Each” skeletons generates five instruments, one for each value instance. This is shown in the left side of Figure 2 on page 15. One type “Each” instrument is defined in the left side of the console and three processes were selected to instantiate the skeleton console.

Again, one console may define more than one skeleton instrument. You can define consoles with both skeleton instrument types while any non-skeleton instruments in the same console are ignored. The relative placement of the defined instrument is kept unchanged. This may give you small instruments that cannot draw graph data when many value instances are selected, but it is easy to resize the console. If the generated instruments would otherwise become too small, **xmperf** attempts to resize the entire console.

The types of instruments best suited for the “Each” type skeleton instruments are the recording instruments. This is further emphasized by the way instruments are created from the skeleton:

- The relative horizontal placement is never changed.
- The relative vertical position defined by the skeleton is never changed, but is subdivided into the number of instruments to be created.
- Each created instrument has the full width of the skeleton instrument.
- Each created instrument has a height which is the total height of the skeleton divided by the number of value instances selected.

## Wildcard Restrictions

Wildcards must represent a section of a value path name which is not the end point of the path. It could represent any other part of the path, but it only makes sense if that part may vary from time to time or between systems. Currently, the following wildcards make sense:

```
CPU/*/...           Processing units
Disk/*/...          Physical disks
FS/rootvg/*/...     File systems
IP/NetIF/*/...      IP interfaces
LAN/*/...           Network (LAN) interfaces
PagSp/*/...         Page spaces
Proc/*/...          Processes
```

```
hosts/*/...              Remote hosts
Mem/Kmem/*/...           Kernel Memory Allocations
RTime/ARM/xaction/*/... ARM response time and activity
RTime/LAN/*/...          IP response time
```

**Note:** Not all wildcards are available on non-**RS/6000** systems.

The file systems wildcard is one of the two current example of path names where more than one wildcard would be appropriate. It is not uncommon for a system to have more than one volume group defined, in which case you need to define an instrument for each volume group, as follows:

```
FS/rootvg/*/... Root volume group
FS/myvg/*/...   Private volume group
FS/yourvg/*/... Another private volume group
```

The other example is that of ARM response time metrics where the higher level of wildcard is the application identifier and the lower is the transaction identifier.

The **xmperf** program prevents you from specifying multiple wildcards in a skeleton instrument. However, it is possible to use dual wildcards in the **3dmon** program as described in Chapter 6, “3D Monitor,” on page 71.

When a console contains skeleton instruments, all such instruments must use the same wildcard. Mixing wildcards would complicate the selection process beyond the reasonable and the resulting graphical display would be incomprehensible.

## Value Name Display

When all values in an instrument have all or part of the value path name in common, **xmperf** removes the common part of the name from the value names displayed in the instrument and displays the common part in a suitable place. To determine how to do this, **xmperf** examines the names of all values in the containing console.

To illustrate, assume you have a single instrument in a console, and that this instrument contains the values:

```
hosts/birte/PagSp/paging00/%free
hosts/birte/PagSp/hd6/%free
```

Names are checked in the following order:

1. It is checked whether all values in a console have any of the *beginning* of the path name in common. In this case, all values in the console have the part **hosts/birte/PagSp/** in common. Because this string is common for all instruments in the console it can conveniently be moved to the title bar of the window containing the console. It is displayed after the name of the console and enclosed in angle brackets like this:

   ```
   <hosts/birte/PagSp/>
   ```

   The parts of the value names left to be displayed in the instrument are:

   ```
   paging00/%free
   hd6/%free
   ```

2. Each instrument in the console is checked to see if all the value names of the instrument have a common *ending*. In this example, it occurs because both value names end in `/%free`. Consequently, The part of the value names to be displayed in the color of the values is reduced to:

   ```
   paging00
   hd6
   ```

   The common part of the value name (without the separating slash) is displayed within the instrument in reverse video, using the background and foreground colors of the instrument. The actual place used to display the common part depends on the primary graph type of the instrument.

An example of displaying value path names in this manner is shown in Figure 2 on page 15, where the center instrument on the left contains the following values:

```
hosts/nchris/Proc/4569~xmservd/kerncpu
hosts/nchris/Proc/4569~xmservd/usercpu
hosts/nchris/Proc/4569~xmservd/nsignals
```

The process number (4569) is not shown in the instrument because the instrument was configured to only show the name of the executing program.

3. The last type of checking for common parts of the value names is only carried out if the *end* of the names do not have a common part. Using this example, no such checking would be done. When checking is done, it goes like this:

   If the beginning of the value names in an instrument (after having been truncated using the checking described in step 1) have a common part, this string is removed from the value path names and displayed in reverse video within the instrument.

   To illustrate, assume youhave a console with two instruments. The first instrument has the values:

```
hosts/umbra/Mem/Virt/pagein
hosts/umbra/Mem/Virt/pageout
```

   while the second instrument has:

```
hosts/umbra/Mem/Real/%comp
hosts/umbra/Mem/Real/%free
```

The result of applying the three rules to detect common parts of the value names would cause the title bar of the console window to display **<hosts/umbra/Mem/>**. The first instrument would then have the text **Virt** displayed in reverse video and the value names reduced to:

```
pagein
pageout
```

The second instrument would display **Real** in reverse video and use the value names:

```
%comp
%free
```

An example of the previous information can be seen in the Sample xmperf Console figure. The console shown in the figure has three instruments and the values for each instrument come from the same contexts. The path names of the three instruments (clockwise from the top left) are:

```
hosts/nchris/Mem/Real/%free
hosts/nchris/Mem/Real/%comp
hosts/nchris/Mem/Real/%noncomp
hosts/nchris/CPU/cpu0/kern
hosts/nchris/CPU/cpu0/user
hosts/nchris/CPU/cpu0/wait
hosts/nchris/CPU/cpu0/idle
hosts/nchris/Mem/Virt/pagein
hosts/nchris/Mem/Virt/pageout
```

## Hints and Tips for Using Instruments

Certain operations you can perform on an **xmperf** instrument, even legal operations, can produce surprising results. Here are a few of the things that may surprise you:

**Changing primary style**
: It is quite easy to change the primary style of an instrument. However, if you change the primary style from a recording graph to a state graph, the secondary style for all values are changed to that of the new primary style. If you later want to change the instrument back to its original style, then any special secondary styles are forgotten.

  Similarly, secondary style information can be lost if you change from one recording graph style to another. For example, assume an instrument has a primary style of bar graph, three values using this primary style, and a single value using a secondary style, which is line graph. If you change

this instrument's primary style to be line graph, then you lose the information about secondary style. Had you changed the primary style to skyline, then the secondary style would be remembered.

**History and shifting**

The amount of memory used to retain an image of a recording graph is dependent on the size of the history and shifting properties of an instrument. In addition, some versions of the X Window System have a restriction that allows no pixmap (image) to be larger than the largest window that fits on the display. You can easily request the creation of larger pixmaps, by increasing the history or shifting properties.

If the product (history x shift) is too large, the graph is distorted. The only way you can currently change that is to reduce one or both properties.

**Resizing and moving instruments**

When you move or resize an instrument, you see a rubber-band outline of the instrument until you release the mouse button. When you move or resize the instrument so that it overlaps other instruments, the instrument you moved or resized is clipped so as to prevent overlapping instruments. Only when the clipping would result in the window being reduced to less than 6 percent of one of the console's dimensions is the resizing or moving terminated and the instrument reverted to its original size and location.

**Choosing colors**

Whenever you change the color of a value or of the foreground or background of an instrument, you see a palette of available colors displayed. This palette might obscure the instrument where you want to change a color. You can move the palette window out of the way by clicking on the title menu bar of the palette window, and holding the button down as you reposition the window.

Notice that when you select a color, the instrument changes immediately. This allows you to experiment with colors without making permanent changes to the instrument. After you select the color you want, select **Proceed** to make the change permanent.

**Using skeleton instruments**

When a configuration file is created on one system and does not use skeleton instruments, differences in machine hardware may make this configuration file less useful on other systems. By using skeleton instruments to make up for such differences, standardized configuration files can be designed and moved between systems.

Some common wildcards are those represented by physical or logical disks, page space on disks, network interfaces, processes, and remote hosts.

**Ghost instruments**

A console designed for one system may contain instruments to monitor values that are not available on another system. If the configuration file for the first system is moved to the second system, and the console is opened, you see empty space in the console where the instrument used to be. The empty space represents what is referred to as a *ghost instrument*.

Ghost instruments occupy the space and prevent you from defining a new instrument in that same space and moving or resizing other instruments to use the space. While this is inconvenient, it serves the purpose of maintaining the console definition intact if you modify other parts of the console. Ghost instruments cannot be removed except by editing the **xmperf** configuration file.

**Small instruments that cannot draw graph data**

If you resize a console so that a recording instrument becomes so small that there is no space in the instrument to draw the graph, graph data may be written into the field at the bottom of the graph usually reserved for time stamps. The data collected in the history buffer is still correct. To correct the display, resize the console to make the instruments large enough to allow graph data to be drawn.

# Consoles

Consoles, like instruments, are rectangular areas on a graphical display. They are created in top-level windows of the OSF/Motif **ApplicationShell** class, which means that if you use the **mwm** window manager, each console has full OSF/Motif window manager decorations. These window decorations allow you to use the **mwm** window manager functions to resize, move, minimize, maximize, and close the console.

## Managing Consoles

Consoles are useful for managing instruments. Some uses for consoles are:

- You can move collections of instruments around in consoles, using the console as a convenient container.
- You can resize a console and still retain the relative size and position of the instruments it contains.
- You can minimize a group of instruments so that historic data is collected and recording of incoming data continues even when the console is not visible. This also helps to minimize the load on your system.
- You can close a console and free all memory structures allocated to the console, including the historic data. Closed consoles use no system resources other than memory to hold the definition of the console.

Consoles can contain non-skeleton instruments or skeleton instruments but not both. Consequently, it makes sense to classify consoles as either non-skeleton or skeleton consoles.

### Non-skeleton Consoles

Non-skeleton consoles can be in either an opened or closed state. Open a console by selecting it from the Monitor menu. After the console is open, it can be minimized, moved, maximized, and resized using **mwm**. None of these actions change the status of the console. You might not see the console on the display, but it is still considered open. If recording has been started, it continues.

If you look at the Monitor menu after you have opened one or more non-skeleton consoles, the name of the console is now preceded by an * (asterisk). This indicates that the console is open. If you select one of the names preceded by an asterisk, you close the corresponding console.

### Skeleton Consoles

Skeleton consoles themselves can never be opened. When you select one from the Monitor menu, you are presented with a list of names matching the wildcard in the value names for the instruments in the skeleton console. If you select one or more from this list, a new non-skeleton console is created and added to the Monitor menu. This new non-skeleton console is automatically opened, and given a name constructed from the skeleton console name suffixed with a sequence number.

The non-skeleton console created from the skeleton is said to be an *instance* of the skeleton console; you say that a non-skeleton console has been *instantiated* from the skeleton. The instantiated non-skeleton console works exactly as any other non-skeleton console, except that changes you make to it never affect the configuration file. You can close the new console and reopen it as often as desired. You can also resize, move, minimize, and maximize the new console.

Each time you select a skeleton console from the Monitor menu you get a new instantiation, each one with a unique name. For each instantiation you'll be prompted to select values for the wildcard, so each instantiation can be different from all others.

If you have created an instance of a skeleton console and you'd like to change it into a non-skeleton console and save it in the configuration file, the easiest way to do so is to select **Copy Console** from the Console menu. This prompts you for a name of the new console and the copy is a non-skeleton console that looks exactly like the instantiated skeleton console from which you copied. After you have copied the console, you can delete the instantiated skeleton console and save the changes in the configuration file.

## Placing Instruments in Consoles

Within their enclosing **ApplicationShell** windows, all consoles are defined as OSF/Motif widgets of the **XmForm** class and the placement of instruments within this container widget is done as relative positioning. Relative positioning has advantages and disadvantages. One advantage is the easy resizing of a console without loss of relative positions of the enclosed instruments. A disadvantage is the complexity involved when adding or removing an instrument in an already full console.

### Adding an Instrument to a Console

When you want to add an instrument to a console, you can choose between adding a new instrument or copying one that's already in the console. If you choose to create an instrument, the following happens:

1. It is checked if there is enough space to create an instrument with a height that is approximately the average height of any existing instruments in the console. If no instruments exist, the height is set to 25% of the console. The space must be available in the entire width of the console. If this is the case, a new instrument is created in the space available.
2. If enough space is unavailable, the existing instruments in the console are resized to provide space for the new instrument. Then the new instrument is created at the bottom of the console. The height of the new instrument will be approximately the average height of any existing instruments, after resizing.
3. If the new instrument has a height less than 100 pixels, the console is resized to allow the new instrument to be 100 pixels high.

If you choose to copy an existing instrument, the following happens:

1. It is checked if there is enough space to create an instrument of the same size as the one you copy. If this is the case, a new instrument is created in the space available. Unlike what happens when adding a new instrument, copying will use space that is just wide enough to contain the new instrument. It's unnecessary to have space available in the full console width.
2. If enough space is unavailable, the existing instruments in the console are resized to provide space for the new instrument. Then the new instrument is created. New space is always created at the bottom of the console, and always in the full width of the console window. However, the new instrument will be the same width as the one from which it was copied.
3. If the new instrument has a height of less than 100 pixels, the console is resized to allow the new instrument to be 100 pixels high.

Rounding may cause the height of the new instrument to deviate 1-2 percent from the intended height.

### Resizing Instruments in a Console

After selecting an instrument and choosing to resize it, the instrument is replaced by a rubber-band outline of the instrument. Resize the instrument by holding the left mouse button down and moving the mouse. When you press the button, the pointer is moved to the lower right corner of the outline. Thus, resizing is always done by moving this corner while the upper left corner of the outline remains stationary.

During resizing, a small button is shown in the top left corner of the rubberband outline. It shows the calculated relative size of the instrument as width x height in percent of the console's total width and height. The relative size is calculated from the relative positions of the edges of the instrument as:

- width = right_edge - left_edge + 1

- height = bottom_edge - top_edge + 1

For example, for an instrument to have a width of 49 and a height of 20, the edges might have the following relative positions:

- top_edge = 20
- left_edge = 1
- right_edge = 49
- bottom_edge = 39

When you release the mouse button the instrument is redrawn in its new size.

Note that it's usually a good idea to move the instrument within the console so that the upper left corner is at the desired position before resizing.

The position of the resized instrument must be rounded so that it can be expressed in percentage of the console size. This can cause the instrument to change size slightly from what the rubber-band outline showed.

Instruments cannot be resized so they overlap other instruments. If this is attempted, the size is reduced so as to eliminate the overlap.

### Moving Instruments in a Console

When you select an instrument to be moved, the instrument disappears and is replaced by a rubber-band outline of the instrument. To begin moving the instrument, place the mouse cursor within the outline and press the left mouse button. Hold the button down while moving the mouse until the outline is where you want it, then release the button to redraw the instrument.

During moving, a small button is shown in the bottom right corner of the rubberband outline. It shows the calculated relative position of the top left corner of the instrument. This helps in positioning the instrument so that it aligns with the other instruments in the console.

Instruments can be moved over other instruments, but are not allowed to overlap them when the mouse button is released. If an overlap would occur, the instrument is truncated to eliminate the overlap.

## The Console Title Bar

The title bar of a console window contains three pieces of information. It might look like this, for example:

`birte: Virtual Memory` *`hosts/xtra/Mem/Virt/`*

The first two pieces of information are always present. The third part is only displayed if all statistics displayed in the console's instruments have some or all of the beginning of their value names in common. The three parts of the title bar text are:

1. The hostname of the system where **xmperf** is executing. It is followed by a colon to separate it from the next part of the text.
2. The name of the console. In case of instances of skeleton consoles, the name might look like this:

   `Disks_#01`

   where the name of the skeleton console would then be *Disks* and the remainder is added to give the instantiated skeleton console a unique name.
3. Any common part of all values displayed in the console enclosed in angle brackets. For a description of how this part is generated, see "Value Name Display" on page 17.

   When values are added to or removed from the console, the common part of the value names might change. When this happens, the console title bar changes to reflect this.

## Environments

Environments are defined in configuration files. By default, **xmperf** reads its environment from the **$HOME/xmperf.cf** file or, if that file does not exist, then as described in Appendix B, “Performance Toolbox for AIX Files,” on page 271. You can override the file name through the command line argument **-o** or the X Window System resource **ConfigFile**. Command line arguments are described in “The xmperf Command Line” on page 29 and supported resources in “The xmperf Resource File” on page 277 section.

In most situations, any person should be able to use a single environment, defining all the consoles required for the monitoring that person needs. However, because the environment holds console definitions and command definitions, as described in Chapter 5, “The xmperf Command Menu Interface,” on page 61, different environments can be defined for different kinds of users. Primarily, this depends on what privilege is required to run the commands.

A system administrator may be authorized to run commands such as **renice** and other commands that require root authority. Therefore, the system administrator may want to have more commands or different ones. When **xmperf** uses such environments, it can be necessary to start the program while logged in as root.

## Monitoring Remote Systems with xmperf

**Note:** This function is only available with the Performance Toolbox Network feature. If you try to access these functions with the Performance Toolbox Local feature only the local hostname is displayed for selection.

Visualizing the load statistics (or monitoring the performance) of a single local host on that same host has been done with a great variety of tools, developed over many years. The tools can be useful for critical hosts such as database servers and file servers, provided you can get access to the host and that the host has capacity to run the tool.

Some of the existing tools, especially when based on the X Window System, allow the actual display of output to take place on another host. Even so, most existing tools depend on the full monitoring program to run on the host to be monitored, no matter where the output is shown. This induces an overhead from the monitoring program on the host to be monitored.

Performance Toolbox for AIX introduces true remote monitoring by reducing the executable program on the system to be monitored to the Agent component’s **xmservd** program, which consists of a data retrieval part and a network interface. It is implemented as a daemon that is started by the **inetd** super-daemon when requests from data consumers are received. The **xmservd** program is described in Chapter 13, “Monitoring Remote Systems,” on page 153.

The obvious advantage of using a daemon is that it minimizes the impact of the monitoring software on the system to be monitored and reduces the amount of network traffic. Because one host can monitor many remote hosts, larger installations may want to use dedicated hosts to monitor many or all other hosts in a network.

The responsibility for supplying data is separated from that of consuming data. Therefore, the term *data-supplier host* is used to describe a host that supplies statistics to another host, while a host receiving, processing, and displaying the statistics is called a data-consumer host.

### The Meaning of Localhost in xmperf

All data-consumer programs made to the RSi API (see Chapter 19, “Remote Statistics Interface Programming Guide”), such as **xmperf**, are always doing remote monitoring in the sense that they can get their flow of statistics only from data supplier daemons. It is immaterial to the protocol, as to the programs, whether the daemon feeding a particular instrument runs on the local or a remote host.

It is, however, convenient that you can create and maintain consoles for the local host. The term *Localhost* refers to the host that all instruments in the **xmperf** configuration file are assumed to refer to when no hostname is given as part of their value path names.

The *Localhost* defaults to the host where **xmperf** is executing. Any other host can be selected at the time you start **xmperf**, using the command line argument **-h**. The *Localhost* cannot be changed while **xmperf** is running.

**Note:** A change of *Localhost* has no influence on where commands, defined in the main window pull-down menus, are executed. Commands are always executed on the host where **xmperf** runs.

## When to Identify Data-Suppliers

The **xmperf** program attempts to contact potential suppliers of remote statistics in the following situations:

- When the program starts, it always attempts to identify potential Data-Supplier hosts.
- When five minutes have passed since the last attempt to contact potential data-supplier hosts and the user creates an instrument referencing a remote data-supplier host.
- When five minutes have passed since the last attempt to contact potential data-supplier hosts and the user activates a console containing a remote instrument.
- When five minutes have passed since the last attempt to contact potential data-supplier hosts and the user requests the **Remote Process Window** from the xmperf Utilities menu.
- When the user selects the **Refresh Host List** entry from the xmperf File menu.

The five-minute limit is implemented to make sure that the data-consumer host has an updated list of potential data-supplier hosts. Please note that this is not an unconditional broadcast every five minutes. Rather, the attempt to identify data-supplier hosts is restricted to times where a user wants to initiate remote monitoring and more than five minutes have elapsed since this was last done.

The five-minute limit not only gets information about potential data-supplier hosts that have recently started; it also removes from the list of data suppliers such hosts, which are no longer available. In heavily loaded networks and situations where one or more remote hosts are too busy to respond to invitations immediately, the refresh process may remove hosts from the list even though they do in fact run the **xmservd** daemon. If this happens, use the **-r** command line argument when you invoke **xmperf**. Through this option, you can increase the time **xmperf** waits for remote hosts to respond to invitations.

## How Data-Suppliers are Identified

When **xmperf** is aware of the need to identify potential data-supplier hosts, it uses one or more of the following methods to obtain the network address for sending an invitational **are_you_there** message. For a full description of network packet types and the network protocol see “The xmquery Network Protocol” on page 159. The last two methods depend on the presence of the file **$HOME/Rsi.hosts**. See Appendix B, “Performance Toolbox for AIX Files,” on page 271 for alternative locations of the **Rsi.hosts** file. The three ways to invite data-supplier hosts are:

1. Unless instructed not to by you, **xmperf** finds the broadcast address corresponding to each of the network interfaces of the host where **xmperf** is executing. The invitational message is sent on each network interface using the corresponding broadcast address. Broadcasts are not attempted on the Localhost (loopback) interface or on point-to-point interfaces such as X.25 or SLIP (Serial Line Interface Protocol) connections.
2. If a list of Internet broadcast addresses is supplied in the file **$HOME/Rsi.hosts**, an invitational message is sent on each such broadcast address. Every Internet Protocol (IP) address given in the file is assumed to be a broadcast address if its last component is the number 255. Note that if you specify the broadcast address of a local interface, broadcasts are sent twice on those interfaces. In large networks, this may produce an unacceptably large number of response to invitational packets.
3. If a list of hostnames or non-broadcast IP addresses is supplied in the file **$HOME/Rsi.hosts**, the host IP address for each host in the list is looked up and a message is sent to each host. The look-up is

done through a **gethostbyname()** call, so that whichever name service is active for the host where **xmperf** runs is used to find the host address. If your nameserver is remote or often slow to respond, specify IP addresses rather than hostnames to avoid the delay caused by the name lookup.

The file **$HOME/Rsi.hosts** has a simple layout. Only one keyword is recognized and only if placed in column one of a line. That keyword is: **nobroadcast**, and means that the **are_you_there** message should not be broadcast using method 1 (where an invitation is sent to the broadcast address of each network interface on the host). This keyword is useful in situations where there is a large number of hosts on the network and only a well-defined subset should be remotely monitored. To say that you don't want broadcasts but want direct contact to three hosts, your **$HOME/Rsi.hosts** file might look like this:

```
nobroadcast
birte.austin.ibm.com
gatea.almaden.ibm.com
umbra
```

The previous example shows that the hosts to monitor do not necessarily have to be in the same domain or on a local network. However, doing remote monitoring across a low-speed communications line is not likely to make you popular with other users of that communication line.

Be aware that whenever you want to monitor remote hosts that are not on the same subnet as the data-consumer host, you must specify the broadcast address of the other subnets or all the host names of those hosts in the **$HOME/Rsi.hosts** file. The reason is that IP broadcasts do not propagate through IP routers or gateways.

**Note:** Other routers can be configured to disallow UDP broadcast between subnets. If your routers disallow UDP broadcasts, type the IP address or hostname of all the hosts you want to monitor on other subnets in the **$HOME/Rsi.hosts** file.

The following example illustrates a situation where you want to do broadcasting on all local interfaces, want to broadcast on the subnet identified by the broadcast address 129.49.143.255, and also want to invite the host called **umbra**:

```
129.49.143.255
umbra
```

**Note:** The subnet mask corresponding to the broadcast address in this example is 255.255.240.0 and the range of addresses covered by the broadcast address is 129.49.128.0 through 129.49.143.255.

## Requesting Exception Messages

One of the message types passing between dynamic data-supplier and data-consumer hosts has a field that is used to tell the responding **xmservd** daemons whether any exception notifications (actually network packets of type **except_rec**) they may generate should be sent to the data-consumer host. Application programs control this field through the last argument to the **RSiOpen** and **RSiInvite** subroutine calls of the Remote Statistics Interface API. By default, the **xmperf** program does not request exception messages to be sent to it. This can be controlled through the command line argument **-x** or the X resource **GetExceptions**. Exception messages are used to inform about abnormal conditions detected on a system. They are described in the "Handling Exceptions" on page 159. When **xmperf** receives an exception message, it is displayed in the **xmperf** main window. No other action is taken. A better way of monitoring exceptions is provided by the program **exmon** described in the Chapter 8, "Monitoring Exceptions with exmon," on page 85.

## Remote Processes

Two of the three main window menus that are used to define command menus have a fixed menu item. Those main window menus are Controls and Utilities. This section describes the **Remote Processes** fixed

menu item in the Utilities pull-down menu. The purpose of the Remote Processes menu item is to provide you with an easy way to display the CPU-intensive processes on a remote host, which runs the **xmservd** daemon.

If you are monitoring remote systems you will recognize the need for this function. It is not uncommon that a console used to monitor a remote host suddenly shows that something unexpected or unusual is happening on the host. To see what causes this, you would need to look into the processes that run on the remote host. However, it takes time to do a remote login and may even be impossible for certain types of errors or certain types of loads. This menu allows you to list key data for all processes running on the remote host without the need to do a remote login.

When you select **Remote Processes** you immediately see a list of remote hosts from which to select. A selection list looks similar to the following *Host Selection List from xmperf* example:

```
beany           9.3.84.132
bobcat          9.3.84.143
drperf          9.3.84.240
eel             9.3.84.160
ender           9.3.84.162
hpserv          9.3.70.236
jaboni          9.3.84.183
jasmine         9.3.84.236
lighting        9.3.84.186
oilers          9.3.67.39 **
perfhp          9.3.70.227
rollie          9.3.84.238
trigger         9.3.84.225
turbojet        9.3.84.228 **
```

## Remote Process List

When you select a host to monitor from **Remote Processes**, you immediately see a list of running processes in the remote host at the time you made the selection. It depends on the currently active display option (see “Remote Processes Menu” on page 27 for details of how to set this option) of the remote process list whether all the processes of the remote host are shown, or whether only CPU-active processes are included. The list shows the most interesting details about the processes, and is sorted in descending order according to the CPU percentage used by the process. The following example shows how a *Remote Process List from xmperf* may be displayed:

```
hosts/trigger:  Process xlcentry  (11644) %cpu 87.0, PgSp: 6.5mb, uid nchris
hosts/trigger:  Process wait      (516)   %cpu  5.8, PgSp: 0.0mb, uid root
hosts/trigger:  Process xlC       (3450)  %cpu  1.5, PgSp: 0.0mb, uid nchris
hosts/trigger:  Process xmservd   (13146) %cpu  0.6, PgSp: 0.3mb, uid root
hosts/trigger:  Process xmperf    (12312) %cpu  0.6, PgSp: 1.5mb, uid nchris
hosts/trigger:  Process make      (13546) %cpu  0.1, PgSp: 0.3mb, uid nchris
hosts/trigger:  Process           (0)     %cpu  0.0, PgSp: 0.0mb, uid root
hosts/trigger:  Process syncd     (4178)  %cpu  0.0, PgSp: 0.0mb, uid root
hosts/trigger:  Process inetd     (6120)  %cpu  0.0, PgSp: 0.1mb, uid root
hosts/trigger:  Process AIXPowerM (3796)  %cpu  0.0, PgSp: 0.5mb, uid root
hosts/trigger:  Process portmap   (5854)  %cpu  0.0, PgSp: 0.2mb, uid root
```

The fields in the list are, from left to right:

**Host path**
The path used to get to the remote host in the form *hosts/hostname*.

**Command Name**
The text “Process” followed by the (first 9 bytes of) the command that executes in the process.

**Process ID**
The process ID (PID) of the process.

**Latest CPU Percentage**
The first time the remote process list is created after the start of the **xmservd** daemon on the remote host, this field shows the CPU usage of the process over its life time. The

same is true whenever a new process shows up on the process list after a refresh. In all other cases, this field shows the average CPU usage since the last refresh of the process list.

**Page Space in Megabytes**
The number of megabytes currently allocated for paging space on an external disk for this process.

**Effect User-ID**
The effective user ID for the process, as changed by **setuid** or **su** if applicable.

**Note:** The fields shown in remote process lists for non-**RS/6000** systems may vary from the fields shown here.

## Remote Processes Menu

When the process overview list is displayed, a menu bar is available to control the list. The following menu items are available:

**File** This menu item yields a pull-down menu with four menu items:

**Refresh** Refreshes the list by reading the current process information from the daemon on the remote host.

**Show All** Changes the display option of the remote process list so that all processes of the remote host are shown, regardless of their CPU usage.

**Show CPU Active**
Changes the display option of the remote process list so that only processes of the remote host that have been using CPU since the last refresh are shown.

**Close** Closes the list.

**Help** Displays any help text supplied in the simple help file and identified by the name Remote Process List.

**Note:** The process list is not updated by **xmperf** automatically. It is your responsibility to use the **Refresh** menu item to update the list as needed. It is updated whenever you change the display option.

# Chapter 3. The xmperf User Interface

This chapter provides information about the **xmperf** user interface.

## The xmperf User Interface Overview

The **xmperf** program has one main window that is displayed when you start the program. The main window provides you with the interface to functions that are not related to active consoles. In addition to the main window, one or more consoles might be displayed when you start **xmperf**. This happens if you have defined one or more default consoles in your configuration file. If no default console is defined, initially only the main window is displayed. When default consoles are defined, they are opened in the same sequence as they are displayed in the configuration file. See "Defining Default Consoles" on page 277 for further information.

## The xmperf Command Line

The general format of the **xmperf** command line is:

**xmperf** [**-v auxz**] [**-w** *width*] [**-o** *options_file*] [**-p** *weight*] [**-h** *localhostname*]
[**-r** *network_timeout*]

All command line options are optional and all except **-r** and **-h** correspond to X Window System resources that can be used in place of the command line arguments. The options **v**, **a**, **u**, **x**, and **z** are True or False options. If one of those options is set through an X Window System resource, it cannot be overridden by the corresponding command line argument. More information on the options can be found in "Execution Control Resources" on page 280. The options are described as follows:

**-v**
: Verbose. This option prints the configuration file lines to the **xmperf** log file `$HOME/xmperf.log` as they are processed. Any errors detected for a line will be printed immediately below the line. The option is intended as a help to find and correct errors in a configuration file. Use the option to understand why a line in your configuration file does not have the expected effect.

  Setting the X Window System resource **BeVerbose** to `True` has the same effect as this flag.

**a**
: Adjust size of the value path name that is displayed in instruments to what is required for the longest path name in each instrument. The length can be less than the default fixed length (or the length specified by the **-w** option if used) but never longer. The use of this option can result in consoles where the time scales are not aligned from one instrument to the next.

  **Note:** For pie chart graphs, adjustment is always done, regardless of this command line argument. Setting the X Window System resource **LegendAdjust** to `True` has the same effect as this flag.

**u**
: Use popup menus. As described in "Console Windows" on page 33, the overall menu structure can be based upon pull-down menus (which is the default) or popup menus as activated with this flag. Typically, pull-down menus are easier to understand for occasional users; while popup menus provide a faster, but less intuitive interface.

  Setting the X Window System resource **PopupMenus** to `True` has the same effect as this flag.

**x**
: Subscribe to exception packets from remote hosts. This option makes **xmperf** inform all the remote hosts it identifies that they should forward exception packets produced by the **filtd** daemon, if the daemon is running. If this flag is omitted, **xmperf** will not subscribe to exception packets.

  Setting the X Window System resource **GetExceptions** to `True` has the same effect as this flag.

**z**
: For monochrome displays and X stations, you might want to try the **-z** option, that causes **xmperf**

© Copyright IBM Corp. 1994, 2004 

to draw graphical output directly to the display rather than always redrawing from a pixmap. By default, **xmperf** first draws graphical output to a pixmap and then, when all changes are done, moves the pixmap to the display. Generally, with a locally-attached color display, performance is better when graphical output is redrawn from pixmaps. Also, a flaw in some levels of X Window System can be bypassed when this option is in effect.

Setting the X Window System resource **DirectDraw** to `True` has the same effect as this flag.

**w** Must be followed by a number between 8 and 32 to define the number of characters from the value path name to display in instruments. The default number of characters is 12.

Alternatively, the legend width can be set through the X Window System resource **LegendWidth**.

**o** Must be followed by a file name of a configuration file (environment) to be used in this execution of **xmperf**. If this option is omitted, the configuration file name is assumed to be **$HOME/xmperf.cf**. If this file is not found, the file is searched for as described in Appendix B, “Performance Toolbox for AIX Files,” on page 271.

Alternatively, the configuration file name can be set through the X Window System resource **ConfigFile**.

**p** If given, this flag must be followed by a number in the range 25-100. When specified, this flag turns on averaging or weighting of all observations for state graphs before they are plotted. The number is taken as the *weight percentage* to use when averaging the values plotted in state graphs. The formula used to calculate the average is:

```
val = new * weight/100 + old * (100-weight) / 100
```

where:

*val* Is the value used to plot.

*new* Is the latest observation value.

*old* Is the *val* calculated for the previous observation.

*weight* Is the weight specified by the **-p** flag. If a number outside the valid range is specified, a value of 50 is used. If this flag is omitted, averaging is not used.

Alternatively, the averaging weight can be set through the X Window System resource **Averaging** (see “Execution Control Resources” on page 280).

The weight also controls the calculation of weighted average in tabulating windows.

**h** Must be followed by the host name of a remote host that is to be regarded as *Localhost*. The *Localhost* is used to qualify all value path names that do not have a host name specified. If not specified, *Localhost* defaults to the host where **xmperf** executes.

**Note:** With the Performance Toolbox Local feature, this flag always uses the local host name.

**r** Specifies the timeout (in milliseconds) used when waiting for responses from remote hosts. The value specified must be between 5 and 10,000. If not specified, this value defaults to 100 milliseconds.

**Note:** On networks that extend over several routers, gateways, or bridges, the default value is likely to be too low.

One indication of a too low timeout value is when the list of hosts displayed by **xmperf** contains many host names that are followed by two asterisks. The two asterisks indicate that the host did not respond to **xmperf** broadcasts within the expected timeout period. The Host Selection List from xmperf (see “Remote Processes” on page 25) shows how some hosts in a host selection list have asterisks. The list shown was generated in a network with multiple levels of routers where the default timeout is on the low side during busy hours.

## The xmperf Main Window

The xmperf main window is shown in the following figure. At the top of the main window is a menu bar. The menu bar provides access to six pull-down menus. The remainder of the window displays messages from **xmperf** as necessary. Message lines in the main window can be scrolled horizontally if they are longer than the window is wide, and you can scroll vertically to see previous message lines. The messages you see could be:

**Information messages**
: Messages from **xmperf** during startup and about commands executed from one of the following menus, including the exact command line the system attempted to execute.

**Error messages**
: Messages telling you about errors that could not be detected during startup.

**Exception messages**
: Exceptions received from data suppliers.

The **xmperf** Main Window is displayed similar to the following example:

```
Building the color table...
Parsing configuration file...
Xmperf Version 2.2 for AIX.
Xmquery Protocol Version 02.03
Xmperf initialized - use "Monitor" menu to open consoles.
Initial console "Mini Monitor" being opened.
```

The menu bar of the main window provides major ways to control **xmperf**. It has the following pull-down menus:

**File** The CUA prescribed File menu.

**Monitor**
: The menu from where you open and close consoles, instantiate skeleton consoles, and create new consoles.

**Analysis**
: One of three tools menus from where commands can be executed. The menu is customizable and is intended to be used for analytical tools.

**Controls**
: The second of three tools menus from where commands can be executed. This menu has a fixed menu item that creates a list of processes in the local system. The rest of the menu is fully customizable and is intended to be used for commands that influence, or control, the performance of your system.

**Utilities**
: The last of three tools menus from where commands can be executed. This menu has a fixed menu item to create a list of processes on a remote host that you must select from a pull-down menu. The rest of the menu is fully customizable and is intended to be used for miscellaneous tools and commands.

**Help** The Help menu.

## The File Menu

The **xmperf** File menu has the following items:

**Save All Changes**
: When you select this menu item, all changes to all consoles are written to the configuration file. The only exceptions are changes to instances of skeleton consoles. Changes to instantiated skeleton consoles are never written to the configuration file, because such consoles don’t belong in the file.

After all changes are saved, the menu selection is inactivated and cannot be selected until new changes are made to at least one console.

**Playback**
This menu selection starts the playback feature described in Chapter 4, “Recording and Playback with xmperf,” on page 53.

**Refresh Host List**
Usually, the remote interface makes sure you have an updated list of remote hosts to select from when you instantiate a remote skeleton console or create a list of processes on a remote host. This automatic refresh is done once every five minutes. The Refresh Host List menu item allows you to initiate a refresh whenever you like.

**Exit xmperf**
Selecting this item terminates the entire **xmperf** application, but not before a couple of checks are done:

1. The program checks whether any changes have been made to any console since the last saving to the configuration file. If so, a small dialog box is displayed, giving you the choice between discarding the changes or saving them to the configuration file.
2. The program checks if any consoles are still active. If this is the case, you are asked if you want to exit while consoles are active. This is just a precaution to prevent accidental exiting of the program when valuable historic data is accumulated in an active console. You can choose to exit or not.

**Note:** An attempt to close the main window from the window manager has the same effect as selecting **Exit xmperf** from the File menu.

## The Monitor Menu

The Monitor menu has four menu items, one of which is representing a submenu:

**Instantiate Skeleton**
Represents a submenu containing all the skeleton consoles defined in the configuration file. To instantiate a particular skeleton console, click on it using the left mouse button. Notice that the skeleton console names are preceded by the letter *S*. For information on how skeleton consoles work, see “Skeleton Consoles” on page 20.

**Add New Console**
Select this option if you want to build a new console interactively from scratch. Details on this selection are provided in “Creating a Console” on page 47.

**Ordinary Consoles**
Consoles defined in the configuration file or created interactively are each represented by one menu selection. When a console is active, the name of the console is preceded by an asterisk. To activate (open) a console, select one with no asterisk; to deactivate (close) a console, select one with an asterisk. Observe that when you close a console, all historic data accumulated in any instrument of that console is lost and the recording of data is stopped.

**Instantiated Skeleton Consoles**
These consoles are also displayed in the Monitor menu. When a skeleton console is selected from the Instantiate Skeleton submenu, an instance of that skeleton is created and activated (opened). This causes it to be displayed in the Monitor menu with an asterisk. From this point, the instance works exactly as an ordinary console except that the changes you make to the instance are not saved to the configuration file.

## The Tools Menus

The Tools menus are described in detail in Chapter 5, “The xmperf Command Menu Interface,” on page 61. Briefly, they allow you to execute commands from within the **xmperf** application with an easy way to fill-in command line arguments. Commands are defined in the configuration file.

## The Help Menu

This pull-down menu has four menu items:

**Help on Main Window**
Displays any help text supplied in the simple help file and identified by the name Main Window.

**Help on Help**
Displays any help text supplied in the simple help file and identified by the name Help on Help.

**Help Index**
Opens a Help Index window with a list of help topics. To display the help screen for a help topic, select the corresponding line in the help index window.

**On Version...**
Displays an information window that states the **xmperf** version in use.

## Console Windows

To give you full flexibility, consoles can be opened and closed from menus associated with each console as well as from the main window. Equally important, consoles can be configured interactively from the same menus. This necessitates a rather intricate system of menus, which within the CUA specifications can be created as either pull-down or popup menus.

By default, **xmperf** is configured with pull-down menus. If you want popup menus, use the command line argument **-u** or set the X Window System resource **PopupMenus** (see “Execution Control Resources” on page 280) to True. Command line arguments are described in “The xmperf Command Line” on page 29 and supported resources in “The xmperf Resource File” on page 277.

### Console Pull-down Menus

When **xmperf** is configured with pull-down menus, each console has its own menu bar. The menu bar has five pull-down menus:

**File** The CUA prescribed File menu.

**Edit Console**
A menu used to customize the console and the instruments it contains, not individual values.

**Edit Value**
A menu for customizing individual values to be plotted by the console’s instruments.

**Recording**
The menu used to start and stop recording from a console or one or more of its instruments.

**Help** The Help menu for the console.

Most choices from the Edit Console and Edit Value menus require that you specify the instruments in the console with which you want to work. Because of this, those menu items are ghosted or unavailable until you have selected an instrument.

When an instrument is selected, **xmperf** attempts to draw a dashed line around it. This is only possible if there is space between the instrument and neighboring instruments or the console border. Ususally you can see some of the dashed line. To always see the dashed line, ensure that the instruments never touch each other or the border of the console which contains it.

After selecting an instrument, inactive menu items in the Edit Console and Edit Value pull-down menus become active, and any selection that you make applies to the selected instrument.

## The Console File Menu

The File menu of a console contains functions that apply to the console or to the **xmperf** application as a whole. The menu items are:

**Save Changes**
When the console has not been modified by you, or when such modifications have already been saved previously, this menu selection is inactive so you can't select it. When you make a change to the console, this menu selection becomes active.

When you select this menu item, all changes to the console (but not to other consoles) are written to the configuration file. After the changes are saved, this menu selection is inactivated until new changes are made to the console.

**Copy Console**
When you select this option, a new console is made as an exact copy of the current console. First, a dialog box prompts you for a name for the new console. You enter the name of your choice as described in "Choosing a Name" on page 47. All the rest is done automatically.

**New Console Path**
This menu item gives you the possibility to "remount" all the instruments in the console on a different host. The name of the menu item means that you replace the *hosts* part of the path names of all values in the console with a new host name. For example, assume a console has two instruments, one monitoring statistics on host **umbra** and the other monitoring statistics on host **bamse**. By selecting a new host name, such as **buzzer**, you cause both instruments to be monitoring **buzzer**.

The new host name is selected from a popup list of host names containing all the currently available data-supplier hosts. From this list, pick the one you want by selecting it and then selecting the Done menu in the menu bar of the box. This produces a small pull-down menu. If you select the **Cancel** menu item, the box goes away, and no new path is selected. If you select the **Reselect** menu item, the list of data-supplier hosts is refreshed. If you select the **Accept Selection** menu item, the selected host is chosen as the remote host for all statistics in the console.

**Note:** With the Performance Toolbox Local feature, only the local host is available for selection.

When the instruments are changed to monitor the new host, some may reference statistics that are not available on the new host. Such statistics disappear from the changed instrument. Similarly, an instrument can contain referenced statistics that were not available on the previous host but exist on the new host. Such statistics are added to the instrument.

**Open Console**
This selection produces a popup menu containing all the consoles defined. The popup menu contains all the consoles in the Monitor menu of the main window and its submenu of skeleton consoles. It is placed here to allow you total control even without having the main window visible.

**Close Console**
When you select this item, the current console is closed and all historic data collected for its instruments is lost. If recording was active for the console or any of its instruments, recording stops and the recording file is closed. You can get the same effect by selecting the console from the Monitor menu of the main window or from the Open Console menu. Because the console is active, its name is preceded by an asterisk in the menus, so selecting it deactivates (closes) the console. An alternative way of closing the console is to select the **Close** option from the Window Manager menu of the console window.

**Erase Console**
Selecting this item erases the console definition from the Monitor menu (and from the configuration file if and when the changes are saved). Before the console is erased the actions described for the

**Close Console** selection are carried out. To make sure you don't do this accidentally, you are prompted to verify the selection before it is carried out.

**Exit xmperf**
This item works exactly like the **Exit xmperf** selection of the File menu in the main window. It is placed here to allow you total control even when the main window is minimized.

**Help** This item contains three menu lines. The first menu line provides you with the intended use of the console and related other consoles and tools. Help for the console is shown if the simple help file (see "Simple Help File Format" on page 286) is present and contains a help screen for the console. The second takes you to the help index, and the last is the prescribed **On Version** to display a short message informing you of the version of **xmperf**.

The **Help** menu item in the File menu is a duplicate of the **Help** menu item in the console menu bar.

## The Edit Console Menu

The pull-down menu for Edit Console contains ten items. **Add Local Instrument** and **Add Remote Instrument** are always active. **Select Instrument** is active only when no instruments are selected. The remaining seven items are active only when an instrument is selected; these menu items are as follows:

**Add Local Instrument**
Selecting this menu item causes the console to be prepared for the addition of a new instrument. Space is acquired in the console as described in "Adding an Instrument to a Console" on page 21. Initially, the instrument is not created. Instead, you are presented with a list of values from which to select the first value of the instrument. The list allows you to select any value on Localhost (see "The Meaning of Localhost in xmperf" on page 23). If your first action is to select **End Selection** in the selection box, the instrument is not created.

When you've selected a value for your instrument the instrument is created with that value as its first one. You then see a dialog box that allows you to select the way you want this value to be plotted. For details on this, see "Changing the Properties of a Value" on page 48. When you have set the options for the first value of the new instrument, you can select and set the options of additional values to be added to the instrument. When you're done, select **End Selection** in the selection box.

**Add Remote Instrument**
Every instrument must show values from one single data-supplier host. If the instrument shows values from a remote data-supplier host, it is called a remote instrument. This menu selection allows you to add a remote instrument.

The first you'll see when you select **Add Remote Instrument** is a selection box with a list of all the currently available data-supplier hosts. From this list, pick the one you want by selecting it and then selecting the Done menu item in the menu bar of the box. This produces a small pull-down menu. If you select the **Cancel** menu item, the box goes away and no instrument is created. If you select the **Reselect** menu item, the list of data-supplier hosts is refreshed. If you select the **Accept Selection** menu item, things proceed as previously described for Add Local Instrument, except that you are presented with a list of values on the data-supplier host you selected.

**Note:** With the Performance Toolbox Local feature of Version 2.2 or later, only the local host is available for selection.

**Tabulating Window**
Select this menu item to display a tabulating window for the selected instrument. If a tabulating window is already displayed when you select this item, that window is closed. *Tabulating windows* are special forms of windows that tabulate the values of the instrument as data is received and will also display a line with a weighted average for each value. Tabulating windows are described in more detail in "Tabulating Windows" on page 50

**Copy Instrument**
Causes a new instrument to be added to the console. The new instrument is an exact copy of the currently selected instrument and is added as described in "Adding an Instrument to a Console" on page 21.

**Delete Instrument**
As a precaution against unintended deletion, a dialog box opens when you ask to delete an instrument. You then have to accept the deletion or cancel it.

**Modify Instrument**
This selection causes a cascade menu to open. The cascade menu has ten items, all of which are concerned with the status and properties (as described in "Instruments" on page 12) of the instrument rather than the properties of individual values in the instrument. The menu items are:

- Resynchronize
- Autoscale
- Interval
- History
- Shift
- Space
- Style & Stacking
- Foreground
- Background
- Change Path

All these are described in the "Modify Instrument Submenu" on page 43.

**Resize Instrument**
Allows you to resize the selected instrument as described in "Resizing Instruments in a Console" on page 21.

**Move Instrument**
Allows you to move the selected instrument as described in "Moving Instruments in a Console" on page 22.

**Unselect Instrument**
Deselects the selected instrument and removes the dashed line around it.

**Select Instrument**
Serves as a reminder of how you select instruments. When you select this item an information window opens and gives you a brief description of the selection principles.

## The Modify Instrument Submenu

This submenu opens when you select the **Modify Instrument** menu item from the Edit Console menu. It contains menu items to modify all of the properties that apply to an instrument. The items are:

**Resynchronize**
Allows you to ask for the instrument's network connection to the data-supplier host to be resynchronized (renegotiated). Make this selection when you notice that the instrument is no longer receiving input, which indicates that the remote supplier host is no longer on the network or that its **xmservd** daemon has aborted or was killed. If you select this menu item and the instrument is still not receiving input from the data-supplier host, most likely the remote host is not up. Wait until you can get a response to a ping, before trying again.

If, after having resynchronized one instrument, **xmperf** estimates the total resynchronizing to take more than 12 seconds, a dialog box opens. From the window you can select to terminate the

resynchronizing or continue it. If you terminate it, none or only some of the instruments defined for the remote host will be active. Under usual circumstances, at least one instrument should be active.

If the estimated total elapsed time to complete the operation increases greater than 120% of what you previously approved (during any one resynchronizing operation), the dialog box opens again showing you the new estimated time to completion of the operation.

**Note:** While this dialog box is displayed, no other window can be used. You must select either **Continue Resync** or **Stop Resync** to remove the dialog box before other X events are processed.

**Autoscale**

Sometimes an instrument receives data values that are far greater than the high range of the instrument. Because recording type instruments can show no more than 105 percent of the high range (scale), readings greater than 105 percent are truncated at approximately 105 percent.

To find the right scale in such situations, use this menu selection. When you select it, **xmperf** scans all values in the instrument to determine if any one exceeds 105 percent of the high scale. The scan uses all data values collected in the history of the instrument. Any value that does exceed 105 percent of the high scale at any point in the recorded history has its high scale adjusted so that the highest peak is shown somewhere between the 50% and the 100% mark in the graph.

If stacking is in effect for the instrument, the peak is determined as the sum of all values using the primary style of the graph at any one point in history.

The changed scales are recorded with the instrument as if you had made the change manually. Therefore, when you leave **xmperf**, you'll be asked whether you want to save the changes to the configuration file.

**Interval**

Opens a dialog box and displays a sliding scale with the current value of the interval property of the instrument. The sliding scale adjusts to the current value so that you can change small values with a granularity of 0.1 second and larger values with a granularity of one minute. By using the slider, you can set the sampling interval in the range 0.2 second to 30 minutes. To change the interval between observations, select the slider with the mouse and move it to the value you want. Then release the slider and select **Proceed**. The **xmservd** daemon on the remote Data-Supplier host is sent a **change_feeding** message every time you select **Proceed**.

Even though the sampling interval can be requested as any value in the previous range, it may be rounded by the **xmservd** daemon on the remote system that supplies the statistics. For example, if you request a sampling interval of 0.2 second but the remote host's daemon is configured to send data no faster than every 500 milliseconds, then the remote host determines the speed.

When the interval is changed, the instrument is redrawn with the new properties including a new pixmap (image) of the instrument if it is a recording instrument. Note that until a time corresponding to the size of the history property has elapsed, the history of the instrument is a mixture of observations taken with the old interval and the new one you chose.

If you select the **Save Buffer** option from one of the recording menus, the data saved to the recording file will have time stamps that assume the interval has been unchanged (and identical to the value of the interval property at the time the buffer is saved). It is suggested that you don't save buffers of instruments that have had their sampling interval changed if exact timing of historic events is important.

**History**

Opens a dialog box and displays a sliding scale, with the current value of the history property of the instrument. The scale ranges from 50 to 5,000 observations. To change the number of observations, select the slider with the mouse and move it to the value you want. Then release the slider, and select **Proceed**.

When the history is changed, the instrument is redrawn with the new properties including a new pixmap (image) of the instrument if it is a recording instrument.

**Note:** If the history property value is reduced, any excess data the instrument collects is discarded; similarly, when the value is increased more memory is allocated to keep the extra history as observations are collected.

**Shift** Opens a dialog box, and displays a sliding scale with the current value of the shift property of the instrument. The scale ranges from one more than the current value of the space property to 20 pixels. To change the number of pixels to shift, select the slider with the mouse and move it to the value you want. Then release the slider, and select **Proceed**.

When the shift property is changed, the instrument is redrawn with the new properties including a new pixmap (image) of the instrument if it is a recording instrument. If you want to reduce the value of this property to a value smaller than the current value of the space property, you must first reduce the value of the space property, then repeat the operation for the shift property.

**Space** Opens a dialog box, and displays a sliding scale with the current value of the space property of the instrument. The scale ranges from zero to one less than the current value of the shift property. To change the number of pixels spacing between bars, select the slider with the mouse and move it to the value you want. Then release the slider, and select **Proceed**.

When the space property is changed, the instrument is redrawn with the new properties including a new pixmap (image) of the instrument if it is a recording instrument. If you want to increase the value of this property to a value that is larger than the current shift value, you must first increase the value of the shift property to one more than what you want the space property to be, then repeat the operation for the space property.

**Style & Stacking**

Opens a dialog box so one of the following style choices can be selected (using radio buttons):

- Line
- Area
- Skyline
- Bar
- State_Bar
- State_Light
- Pie_Chart
- Speedometer

Stacking can also be selected. After selecting style and stacking, select **Proceed**. (Cancel and Help options are also available.)

**Foreground**

Opens a dialog box with one button for each of the colors:

- Black, white, and grey10, grey20, ... grey90.
- The colors defined in the X resource file with the resources **ValueColor1** through **ValueColor24** (see “Resources Defining Default Colors” on page 279).
- Any additional colors referenced in the **xmperf** configuration file.

  Below the color buttons, eleven buttons show how the current selection of foreground and background colors look when each of the eleven tiles is chosen. The eleven tile buttons are numbered for easy reference but in the configuration file, tiles are referred to with symbolic names as described in “Explaining the xmperf Configuration File” on page 272.

**Background**

Works exactly as foreground color, only it changes the background color of the instrument.

**Change Path**

This menu item gives you the possibility to "remount" the instrument on a different host. The name of the menu item means that you exchange the *hosts* part of the path names of all values in the instrument with a new host name. For example, assume the instrument is monitoring statistics on host **pjank**. By selecting a new host name, such as **alvor**, you cause the instrument to be monitoring **alvor**.

The new host name is selected from a popup list of host names containing all the currently available data-supplier hosts. From this list, pick the one you want by selecting it and then selecting the Done menu in the menu bar of the box. This produces a small pull-down menu. If you select the **Cancel** menu item, the box goes away, and no new path is selected. If you select the **Reselect** menu item, the list of data-supplier hosts is refreshed. If you select the **Accept Selection** menu item, the selected host is chosen as the remote host for all statistics in the instrument.

**Note:** With the Performance Toolbox Local feature, only the local host is available for selection.

When the instrument is changed to monitor the new host, it is possible that it references statistics that are unavailable on the new host. Such statistics are not displayed in the changed instrument. Similarly, the instrument can reference statistics that were not available on the previous host, but exist on the new host. Such statistics are added to the instrument.

If recording is active for the instrument at the time you change one of the following properties of the instrument, the change has no effect on the recording. The instrument definition is saved when the recording starts, and subsequent changes do not affect the recording file. The properties influenced are:

- History
- Shift
- Space
- Style
- Stacking
- Foreground
- Background

All this means is that the initial properties of the playback console are as recorded in the recording file. As described in "Playback Console Windows" on page 43, all the previously listed properties can be changed before or during playback.

## The Edit Value Menu

The Edit Value pull-down menu has five items. The first four items are active only when an instrument is selected. The last one is active only when no instrument is selected. The menu items are:

**Add Value**

This menu item is used to add a value to an instrument. It is done by selection from a hierarchy of value selection windows as described in "Value Selection" on page 46.

**Change Value**

When you make this menu selection, you must tell **xmperf** which value you want to change. If the instrument has more than one value, you'll see a dialog box with a set of radio buttons one for each of the values currently defined in the instrument. To continue, select the value you want to change and then on **Proceed**.

When you select a value, or directly if the instrument has only one value, another dialog box opens. You can change any or all of the properties for the value from this dialogue box. This dialog box is described in "Changing the Properties of a Value" on page 48.

**Delete Value**
When you make this menu selection, you must tell the program which value you want to delete from the instrument. To allow this, **xmperf** opens a dialog box with a set of radio buttons one for each of the values currently defined in the instrument. Select the value you want to delete and then on **Proceed**.

When a value is selected, you are asked if you really want to delete the value from the instrument. Select **OK** if you do; otherwise on **Cancel**.

**Unselect Instrument**
Deselects the selected instrument and removes the dashed line around it.

**Select Instrument**
Serves as a reminder of how you select instruments. When you select this item an information window opens and gives you a brief description of the selection principles.

## The Recording Menu

This menu item yields a pull-down menu containing two items. Both items represent cascading submenus. The first item is always active; the second one is active only when an instrument is selected. The menu items are:

**Console Recording**
Opens the recording submenu that allows you to start or stop recording from the entire console. For details, see "Recording Methods" on page 53.

**Instrument Recording**
Opens the recording submenu that allows you to start or stop recording from the selected instrument. For details, see "Recording Methods" on page 53.

## The Help Menu

This item contains two menu lines. The first line provides help on understanding the intended use of the console. Help for the console is shown if a simple help file (see "Simple Help File Format" on page 286) is present and contains a help screen for the console.

The second line is the prescribed **On Version** that displays a short message informing you of the version of **xmperf**.

## Console Popup Menus

When **xmperf** is configured with popup menus, a popup menu opens whenever you move the mouse pointer into a console and click the left or middle mouse button. If the mouse pointer is within the outline of an instrument, that instrument is selected and the full set of pop-up menu choices are active. If the mouse pointer is positioned outside any instrument, the menu is still opens but some of the menu items that require an instrument to be selected are inactive or *ghosted*. Using the right mouse button, select the menu item that you want to select.

The menu that opens contains the following items. Some menu items produce a cascade menu when selected; they are marked with the word *submenu* in parentheses and described in separate sections after the direct menu items:

- Value Editing (submenu)
- Modify Instrument (submenu)
- Add Instrument (submenu)
- Tabulating Window
- Copy Instrument
- Resize Instrument
- Move Instrument

- Delete Instrument
- Console Recording (submenu)
- Instrument Recording (submenu)
- Copy Console
- New Console Path
- Open Console
- Close Console
- Erase Console
- Save Changes
- Exit xmperf
- Help

The xmperf popup menu items that take you directly to a function are the following:

**Tabulating Window**
Select this menu item to display a tabulating window for the selected instrument. If a tabulating window is already displayed when you select this item, that window is closed. Tabulating windows are special forms of windows that will tabulate the values of the instrument as data is received and will also calculate a line with a weighted average for each value. Tabulating windows are described in more detail in "Tabulating Windows" on page 50.

**Copy Instrument**
Causes a new instrument to be added to the console. The new instrument is an exact copy of the currently selected instrument and is added as described in "Adding an Instrument to a Console" on page 21.

**Resize Instrument**
Allows you to resize the selected instrument as described in "Resizing Instruments in a Console" on page 21.

**Move Instrument**
Allows you to move the selected instrument as described in "Moving Instruments in a Console" on page 22.

**Delete Instrument**
A dialog box opens when you ask to delete an instrument. You then accept the deletion or cancel it. This safeguards you against unintended deletion of instruments.

**Copy Console**
When you select this option, a new console is made as an exact copy of the selected console. The first you'll see is a dialog box that prompts you for a name for the new console. You enter the name of your choice as described in "Choosing a Name" on page 47. All the rest is done automatically.

**New Console Path**
This menu item gives you the possibility to "remount" all the instruments in the console on a different host. The name of the menu item means that you replace the *hosts* part of the path names of all values in the console with a new host name. For example, assume a console has two instruments, one monitoring statistics on host **pjotr** and the other monitoring statistics on host **basse**. By selecting a new host name, say **mango**, you cause both instruments to be monitoring **mango**.

The new host name is selected from a popup list of host names containing all the currently available data-supplier hosts. From this list, pick the one you want by selecting it and then selecting the Done menu in the menu bar of the box. This produces a small pull-down menu. If you select the **Cancel** menu item, the box goes away, and no new path is selected. If you select the **Reselect** menu item, the list of data-supplier hosts is refreshed. If you select the **Accept**

**Selection** menu item, the selected host is chosen as the remote host for all statistics in the console. Note that with the Performance Toolbox Local feature, only the local host is available for selection.

As the instruments are changed to monitor the new host, some may reference statistics that are not available on the new host. Such statistics are not displayed in the changed instrument. Similarly, the instrument can contain referenced statistics that were not available on the previous host but exist on the new host. Such statistics are added to the instrument.

**Open Console**
This selection produces a popup menu containing all the consoles defined in the Monitor menu of the main window and its submenu of skeleton consoles. For convenience, this selection is placed here to allow you total control even without having the main window visible.

**Close Console**
When you select this item the current console is closed and all historic data collected for its instruments is lost. If recording is active for the console or any instrument in the console, recording stops and the recording file is closed. You can get the same effect by selecting the console from the Monitor menu of the main window or from the Open Console menu. Because the console is active, its name is preceded by an asterisk in the menus, so selecting it deactivates (or closes) the console. An alternative way of closing the console is to select the **Close** option from the Window Manager menu.

**Erase Console**
Selecting this item erases the console definition from the Monitor menu (and from the configuration file if and when changes are saved). Before the console is erased, the actions described for the **Close Console** menu item are carried out. To make sure you don't delete a console accidentally, you are prompted to verify the selection before it is carried out.

**Save Changes**
When you have not changed the console, or when changes have previously been saved, this menu selection is inactive: you cannot select it. When you make a change to the console, this menu selection becomes active.

When you select this menu item, all changes to the console (but not to other consoles) are written to the configuration file. After the changes are saved, the menu selection is inactivated until new changes are made to the console.

**Exit xmperf**
This item works exactly like the **Exit xmperf** selection of the File menu in the main window. It is placed here to allow you total control even when the main window is minimized.

**Help** This item is intended to provide help to understand the intended use of the console and related other consoles and tools. Help for the console is shown if a simple help file (see "Simple Help File Format" on page 286) is present and contains a help screen for the console.

## Value Editing Submenu

The Value Editing submenu has three items:

**Add Value**
A menu item used to add a value to an instrument. This is done by selection from a hierarchy of value selection windows as described in "Value Selection" on page 46.

**Change Value**
When you make this menu selection, you need to indicate which value you want to change. If the instrument only has one value, the choice is obvious; otherwise you'll see a dialog box with a set of radio buttons, one for each of the values currently defined in the instrument. To continue, select the value you want to change and then on **Proceed**.

When you select a value, another dialog box opens. You can change any or all of the properties for the value from this dialogue box. This dialog box is described in "Changing the Properties of a Value" on page 48.

**Delete Value**

When you make this menu selection, you must indicate which value you want to delete from the instrument. To simplify doing this, a dialog box with radio buttons, corresponding to each of the values currently defined in the instrument, opens. To continue, select the value you want to delete and then select **Proceed**.

When a value is selected you are asked if you really want to delete the value from the instrument. Select **OK** if you do; otherwise select **Cancel**.

## Modify Instrument Submenu

The Modify Instrument submenu is identical whether you have configured **xmperf** for pull-down or popup menus. It is described in "The Modify Instrument Submenu" on page 36.

## Add Instrument Submenu

The Add Instrument submenu has two items:

**Local Instrument**

Selecting this menu item causes the console to be prepared for the addition of a new instrument. Space is acquired in the console as described in "Adding an Instrument to a Console" on page 21. Initially, the instrument is not created. Instead, you are presented with a list of values from which to select the first value of the instrument. The list allows you to select any value on *Localhost* (see "The Meaning of Localhost in xmperf" on page 23). If your first action is to select **End Selection** in the selection box, the instrument is not created.

When you've selected a value for your instrument, the instrument is created with that value as its first one. You will then see a dialog box that allows you to select the way you want this value to be plotted. For details of this, see "Changing the Properties of a Value" on page 48. When you have set the options for the first value of the new instrument, you can select and set the options of additional values to be added to the instrument. When you're done, select **End Selection** in the selection box.

**Remote Host Instrument**

Every instrument must show values from either the host where **xmperf** is executing or from one single data-supplier host. If the instrument shows values from a data-supplier host, it is called a remote instrument. This menu selection allows you to add a remote instrument.

A selection box with a list of all the currently available data-supplier hosts opens. From this list, pick the one you want by selecting it and then selecting the Done menu in the box. This produces a small pull-down menu. If you select the **Cancel** menu item, the box goes away and no instrument is created. If you select the **Refresh Host List** menu item, the list of data-supplier hosts is refreshed. If you select the **Accept Selection** menu item, things proceed as described in Local Instrument (see "Add Instrument Submenu"), except that you are presented with a list of values on the data-supplier host you selected.

**Note:** With the Performance Toolbox Local feature of Version 2.2 or later, only the local host is available for selection.

## Recording Submenus

The **Recording** Submenus **Console Recording** and **Instrument Recording** are identical whether you have configured **xmperf** for pull-down or popup menus. They are described in "The Recording Menu" on page 40.

# Playback Console Windows

Chapter 4, "Recording and Playback with xmperf," on page 53 describes the use of the playback facility of **xmperf**. This section describes how you modify the appearance of a playback console.

Regardless of whether **xmperf** is configured for popup or pull-down menus, the menu you use to change the appearance of a playback instrument is always a popup menu. It is activated by placing the mouse

pointer within an instrument and then clicking the left or middle mouse button. The popup menu you get is a subset of the submenu described in “The Modify Instrument Submenu” on page 36 plus some special items. The menu items are:

- Tabulating Window
- Autoscale
- Maxiscale
- History
- Shift
- Space
- Style and Stacking
- Foreground
- Background
- Change Value
- Delete Value
- Move Instrument
- Resize Instrument
- Erase Instrument
- Annotation Notes
- Write Current View

The things you can do are all related to how the recorded data is presented. You can change any or all of the previously listed properties of the instrument that was clicked on to bring up the menu. This can be done before or after you start playback. All changes, except deletions and the effect of the selections **Autoscale** and **Maxiscale**, can be changed as many times as you like, allowing you to re-play a particular recording with many different appearances.

The following is a brief description of how to use the menu items:

**Tabulating Window**
: Select this menu item to display a tabulating window for the selected instrument. If a tabulating window is already displayed when you select this item, that window is closed. Tabulating windows are special forms of windows that tabulate the values of the instrument as data is received and also calculate a line with a weighted average for each value. Tabulating windows are described in more detail in “Tabulating Windows” on page 50.

**Autoscale**
: When selected, this menu item causes a scan of all data values collected in the history of the instrument. Note that for this scan, only values that have actually been played back since the last **Seek** or **Rewind** are part of the history. Any value that does exceed 105 percent of the high scale at any point in the recorded history has its high scale adjusted so that the highest peak is shown somewhere between the 50% and the 100% mark in the graph.

  If stacking is in effect for the instrument, the peak is determined as the sum of all values using the primary style of the graph at any one point in history.

**Maxiscale**
: When selected, this menu item causes a scan of all data values collected in the history of the instrument. Note that for this scan, only values that have actually been played back since the last **Seek** or **Rewind** are part of the history. All values are adjusted so that the highest peak at any point in the recorded history is shown somewhere between the 50% and the 100% mark in the graph.

  If stacking is in effect for the instrument, the peak is determined as the sum of all values using the primary style of the graph at any one point in history.

**History**
When the history property is changed for a recording instrument, the instrument is redrawn with the new properties including a new pixmap (image) of the instrument. Note that for playback, the purpose of the history property is to define the size of the pixmap (image) kept in memory. This is only useful for recording type instruments and in playback is only used to determine whether you can scroll the displayed graph and by how much.

**Shift** When the shift property is changed for a recording instrument, the instrument is redrawn with the new properties including a new pixmap (image) of the instrument. If you want to reduce the value of this property to a value smaller than the current value of the space property, you must first reduce the value of the space property, then repeat the operation for the shift property.

**Space** When the space property is changed, the instrument is redrawn with the new properties including a new pixmap (image) of the instrument if it is a recording instrument. If you want to increase the value of this property to a value that is larger than the current shift value, you first increase the value of the shift property to one more than what you want the space property to be, then repeat the operation for the space property.

**Style & Stacking**
This selection causes a dialog box to open. The window has a set of radio buttons, one for each possible instrument type, and a single button that allows you to activate or deactivate the stacking facility.

Change the primary style of the instrument by selecting the instrument type you want it to be. Select or deselect stacking using the stacking button. When you've made your selections, select **Proceed** to implement the changes. After the changes are applied to the instrument, it is redrawn from the beginning.

**Foreground**
Allows you to change the foreground color and tile of the instrument.

**Background**
Allows you to change the background color and tile of the instrument.

**Change Value**
When you make this menu selection, you must tell **xmperf** which value you want to change. If the instrument has more than one value, you will see a dialog box with a set of radio buttons one for each of the values currently defined in the instrument. To continue, select the value you want to change and then on **Proceed**.

When you select a value, or directly if the instrument has only one value, another dialog box opens. You can change any or all of the properties for the value from this dialogue box. This dialog box is described in “Changing the Properties of a Value” on page 48. Because no new data values can be added to a recording, the data path cannot be changed from the dialog box.

**Delete Value**
When you make this menu selection, you must tell the program which value you want to delete from the instrument. To allow this, **xmperf** opens a dialog box with radio buttons that correspond to each of the values currently defined in the instrument. Select the value you want to delete and then on **Proceed**.

When a value is selected, you are asked if you really want to delete the value from the instrument. Select **OK** if you do; otherwise select **Cancel**.

**Move Instrument**
Allows you to move the selected instrument as described in “Moving Instruments in a Console” on page 22.

**Resize Instrument**
Allows you to resize the selected instrument as described in “Resizing Instruments in a Console” on page 21.

**Erase Instrument**
: As a precaution against unintended deletion, a dialog box opens when you ask to delete an instrument. You then have to accept the deletion or cancel it.

**Annotation Notes**
: Opens the annotation list window in which any existing annotations are listed and from where new annotations can be added and existing ones modified or deleted.

**Write Current View**
: Modifies the control information in the recording file to reflect the current layout and contents of the instrument. The original control information is not saved. If annotations exist in the recording file, they are reorganized to remove any annotations marked for deletion and inserting the remaining ones in an optimal place in the file.

## Important xmperf Dialogs

This section covers some of the important dialog boxes, or value selection windows, you use to customize **xmperf**.

## Value Selection

Value selection can be invoked from the Add Value menu, implicitly from one of the Add Instrument menus, or from the dialog box used to change the properties of a value as described in “Changing the Properties of a Value” on page 48. Value selection is done from cascading lists that work as follows:

A dialog box (which is called the value selection window) opens, listing the top layer of values from which you can select. In this list, lines ending in a slash and three dots signify that the line itself represents a list at the next hierarchical level. These lines are *context lines*. The remaining lines in the list are called *statistics lines*, each of which represents a value.

The content of the*Value Selection Dialog Box for xmperf* will be similar to the following example. It shows four statistic lines and six context lines.

```
hosts/trigger/FS/rootvg/free        Free space in volume group, MB
hosts/trigger/FS/rootvg/ppsize      Physical partition size
hosts/trigger/FS/rootvg/lvcount     Number fo logical volumes in VG
hosts/trigger/FS/rootvg/pvcount     Number fo physical volumes in VG
hosts/trigger/FS/rootvg/hd4/...     /
hosts/trigger/FS/rootvg/hd2/...     /usr
hosts/trigger/FS/rootvg/hd9var/...  /var
hosts/trigger/FS/rootvg/hd3/...     /tmp
hosts/trigger/FS/rootvg/hd1/...     /home
hosts/trigger/FS/rootvg/lv00/...    /usr/vice/cache
```

If you select a context line, you’re shown the next level of statistics in another dialog box that’s placed just below the top frame of the first dialog box. This new box contains a list that can include context lines as well as statistics lines. You can repeat the process until the last list you get contains only statistics lines. Note that as you move through the hierarchy, the names displayed in the list become longer as the preceding hierarchical levels are prefixed to the names, separated by slashes.

The title bar of each dialog box shows the context path for the lines in that dialog box. This is another way of showing the hierarchical levels of the values.

If you select a statistics line, the result depends on how you invoked this function:

1. If you invoked the function from the Add Value menu or one of the Add Instrument menus, the line you selected immediately is added to the instrument. The value is added at the first empty value slot in the instrument and inherits the color of that slot, whatever that color is. All other properties are initially set to their default values as described in the data structures behind the list you chose from.

Then you see another dialog box (described in "Changing the Properties of a Value" on page 48) that allows you to modify the new value's properties. If you want to add the selected value to the instrument, regardless of whether you changed any of its properties, you need to select **Apply** in the dialog box. If you select **Cancel**, all the changes you made are discarded and the value is removed from the instrument.

When the property change window disappears, you return to the Value Selection window. You can then select another value to be added to the instrument. This can be repeated until the maximum number of values allowed in an instrument is reached. When you have finished adding values, you close the value selection windows by clicking **End Selection**.

2. If you invoked the function from the dialog box used to change the properties of a value (see "Changing the Properties of a Value" on page 48) then the value you selected replaces the value you are modifying. You can repeat the operation (selecting another value), and each time the selected value replaces the instrument value. When you are sure you have the value you want, select **End Selection** in the selection dialog box to return to the property change Window. Now select **Apply** for the change to remain in effect. Select **Cancel** if you want to return to the original instrument definition.

   When you have more than one value selection window displayed you can jump backwards in the *stack* of windows using either of the following two ways:

   **One Level Back**
   This is one of the buttons in the value selection windows. When you click it, the window disappears and the immediately lower window becomes active. This way, you can go one level back before selecting another value.

   **Closing the Window**
   Each Value Selection window has window manager decorations, including the window menu that allows you to close the window. If you close a window this way, you also close all windows on top of that window. If the window you closed is not the lowest window, one or more Value Selection windows remains displayed. This gives you a way of skipping more than one level back before selecting another value. The context path displayed in the window frame helps you determine which window to close.

## Creating a Console

Creating a new console requires two steps:

1. Select a name for the new console and create an empty console.
2. After an empty console is created, add the instruments you want in the new console.

### Choosing a Name

When you select **Add New Console** from the Monitor menu of the main window, a small dialog box opens. The box has an input field where you can type the name you want the new console to have. Initially, the input field has a name constructed from the date and time. You can change the name to anything, if you do not use names of existing consoles.

If the name you enter contains periods (full stops), slashes, or colons, **xmperf** converts these characters to commas, underscores, and semicolons, respectively. This is done to prevent characters in a console name from clashing with the characters used for delimiters in the configuration file or with file names of recording files.

After you have entered the name you want, press the Enter key or select **Proceed** to create the empty new console. The creation of the console causes it to be added to the Monitor menu (with its name preceded by an asterisk because it is active). If the console is empty (has no instruments), it will not be saved to the configuration file even if you ask to save all changes. However, when the new console contains an instrument, it is added to the configuration file when you ask to save it or to save all changes.

## Adding Instruments to the Console

When initially created, the new console is empty. You'll need to add at least one instrument to the console before you can use it for anything. If **xmperf** is configured with pull-down menus, use the Edit Console pull-down menu and select one of the Add Iinstrument menu items. If you use popup menus, select the empty console and select **Add Instrument** from the menu.

# Changing the Properties of a Value

The dialog box used to change the properties of a value opens when you do the following:

- Select **Change Value**.
- Select **Add Value**.
- Add a new instrument.

The window is a true dialog box in the sense that whatever you change from the window has an immediate effect on the instrument and value with which you are working. If the instrument is large or complicated or if your host is heavily loaded, immediate can mean within a few seconds.

The Changing Properties of a Value dialogue box has a check box on the top by the location, and a statement that system-wide time executing in user mode (percent). A check box for Color and a corresponding field to type in the color. Below is a row of radio buttons to indicate one of the following: line, area, skyline, or bar. A field to type in Your Label is below. There are three sliders to set values for the following: lower range for value, upper range for value, and the threshold for value. Below these three values are two radio buttons that indicate the alarm be set either to alarm when above threshold or alarm when below threshold.

Because the effect of your changes are seen as you make them, be sure to start by moving the dialog box so that it doesn't obscure the instrument with which you work. After making all the changes, select **OK** to remove the dialog box and make the changes permanent. Select **Cancel** to restore the original instrument properties.

Three of the properties are represented in the dialog box by a sliding scale. Scales enable you to change a numerical value by using the mouse instead of the keyboard. However, it can sometimes be difficult to hit exactly the value you want. To cope with this problem, the sliders in the dialog box adjust to the value they are displaying. This is done by rounding by 10, 100, 1,000, 10,000, or 100,000. When rounding is done, a multiplication factor is displayed as part of the slider text. Another way to cope with the precision is by limiting the high value to five times the current value. This means that if the current value is 100 and you want to increase it to 100,000, then you must do it in steps. For example:

1. With current value = 100, change to 500.
2. With current value = 500, change to 2,500.
3. With current value = 2,500, change to 10,000.
4. With current value = 10,000, change to 50,000.
5. With current value = 50,000, change to 100,000.

The major advantage is that of not having to accept values such as 99,973 or 100,744 when you want 100,000.

The actual changing of values is done by moving the mouse pointer to the slider and then pressing the left mouse button; while holding the button down, move the mouse pointer left or right to decrease or increase the value. As you move the pointer, the value corresponding to the slider position is displayed. When you have reached the desired value, release the mouse button.

The dialog box always looks the same and contains the following sections:

**Value path and description**
Initially, holds the path name and description of the value you are changing. If you select the button you are presented with the first of a series of value selection windows as described in “Value Selection” on page 46.

As you pick another value for the instrument by selecting from the lists of values, the text displayed here changes to the path name and the description of the new value.

As you change the value, the property values for ranges and threshold also change because the default values corresponding to the new value are taken. Therefore, before changing any other properties, ensure that you are working with the desired value.

**Color** The name of the current color for drawing the value is displayed in a label field next to a button. The label field is painted in the current color and tile of the value.

To change the color and the tile of the value, select the button. This causes the color selection dialog box to open. Select a new color or tile from the dialog box by selecting one of the color selection or tile buttons. Try however many color and tile combinations you want and select **Proceed** in the color selection window when you find what suits you.

**Secondary style**
If the primary style of the instrument you are working with is one of the state graphs, only one radio button is displayed for secondary style. That is because you cannot have a secondary style different from the primary style with state graphs. The single radio button and its text are merely for your information and no action is taken if you select it.

For recording style graphs, however, you always are presented with a radio button for each of the recording graph styles. The one that is used as secondary style for the value you are changing looks as if it is pressed down (selected). You can change the secondary style to any displayed style by selecting the corresponding radio button.

**Your label for the value**
Allows you to specify your own label to be displayed in the instrument for this value. By default, this field is empty, and the path name of the statistic is used in the instrument. By entering a text string of up to 32 characters, you can override the default.

In case of values for processes, you can use two keywords to specify which part of the constructed value name you want displayed as the value name. Processes are identified by a name constructed by concatenating the process ID (PID) with the name of the executing command, separated by a ~ (tilde). If you do not enter your own label, this constructed name is used to identify process values. By entering the keyword **cmd**, you can change this to be only the name of the executing command; by entering the keyword **pid**, you change it to be only the PID of the process. The distributed configuration file has an example of the use of the two keywords in the skeleton console named Local Processes.

**Lower range**
The sliding scale (slider) can move from value zero to one less than the current upper range property value. If you want a non-zero lower scale, first ensure that you have the upper scale set correctly.

**Upper range**
The sliding scale (slider) can move from 1 to 1,000,000,000; but, as described previously, you might have to do this in steps.

**Threshold**
The sliding scale (slider) can move from value 0 to 1,000,000,000; but, as described previously, you might have to do this in steps. The threshold value only has a meaning for state light type instruments. The threshold type (described as follows) determines how to interpret the threshold value.

**Threshold type**
This section has two radio buttons. One, and only one, must be active. The section describes the threshold type used for state light type instruments as either ascending or descending.

## Tabulating Windows

Whenever an instrument is active, it updates a graphical display of the values it contains. Each time a new set of values is received, the graphical window is updated. In addition to the graphical display, an instrument can simultaneously tabulate the data values as they are received. This is done by opening a tabulating window.

A tabulating window is opened for an instrument when you select the menu item **Tabulating Window.** If you do it one more time, the tabulating window is closed. You can also close a tabulating window from its window manager menu.

A tabulating window has three components:
- Header lines
- Weighted average line
- Detail lines

### Tabulating Window Header Lines

The header lines are constructed using either user-supplied labels for the values (if available) or path names. The path name with most levels determines the number of header lines. The first value in the Example of a Tabulating Window figure, has a path name of:

```
hosts/nchris/CPU/cpu0/kern
```

That gives five levels. To keep the number of header lines as low as possible, all tabulating windows show the host name in the upper left corner of the window. Because all values of any instrument have the first two levels of the path name in common, they are not shown for each value.

### Tabulating Window Weighted Average Line

The first data line is the *weighted average line* showing an approximation to an average for each value. The formula used to calculate the average is:

```
new_avg = observation x weight + old_avg (1 - weight)
```

The weight in the formula is specified by the command line argument **-p** or the X resource **Averaging**. The default weight is 50%, is used as 0.5 in the previous formula.

To make the weighted average line stand out from detail lines, it is shown in reverse video.

### Tabulating Window Detail Lines

The detail lines show actual data values as they are received. At the far left of each line is a time stamp showing the time at which the data values were captured, using the time of day of the host that generated the data values. As new observations are received, previous data lines move down and, eventually, disappear off the bottom of the list of detail lines.

The default number of detail lines sent to a tabulating window is 20. However, when the window initially opens, it typically shows only the top four detail lines including the weighted average line. Use the window manager to resize the window if you want to see more detail lines.

The number of detail lines in tabulating windows can be changed with the X resource **TabWindowLines**. You can specify from 2 to 100 lines. The number of lines you specify should include the weighted average line. If you specify more than 25 lines, the tabulating window provides a vertical scroll bar that allows you to scroll down to see the last of the detail lines.

## Column Width in Tabulating Windows

The width of each column in tabulating windows defaults to nine characters. Data values are always cut to fit the active column width. If this would cause digits to be removed from the value, it is divided by either 1,000 or 1,000,000 as required and suffixed with a `K` or `M`, respectively. Where applicable, column headings are truncated so that at least one space separates two headings.

The default column width can be changed through the X resource **TabColumnWidth** (see “Execution Control Resources” on page 280). A value from 5 through 15 can be specified.

## Decimal Places in Tabulating Windows

Some values may be tabulated with one decimal place. This is done if the value’s High Scale property is less than or equal to a global threshold that defaults to 10. This is done to allow extra granularity for values that never have high readings.

The global threshold can be changed with the X resource **DecimalPlaceLimit** (see “Execution Control Resources” on page 280).

## Tabulating Window Title Bar

The title bar of tabulating windows shows the name of the host where **xmperf** is executing followed by `(TAB):` and the identification of the instrument. The instrument is identified by the console name and the sequence number of the instrument within the console.

# Chapter 4. Recording and Playback with xmperf

An active console keeps only as many observations as its history property specifies. When using **xmperf**, you will likely encounter situations where the observations are moving out of your view too fast. For such cases, and to facilitate playing of scenarios collected on other hosts, the Recording and Playback feature of **xmperf** was implemented. Another feature lets you annotate recording files.

In addition to the recording function of **xmperf**, recordings can be created by **3dmon**, the **xmservd** daemon, and by the **azizo** program, and a set of recording support programs. The creation of recordings by these programs are described in:


- Chapter 6, “3D Monitor,” on page 71
- Chapter 9, “Recording Files, Annotation Files, and Recording Support Programs,” on page 93
- Chapter 10, “Analyzing Performance Recordings with azizo,” on page 107.


## Recording of Statistics

Recording of statistics can be initiated for one or more instruments in a console or for all instruments in a console. Recording can be active for more than one console at a time. Recordings are always written to a file, which has a name prefix of “R.”. followed by the name of the console. The file is written to the directory **$HOME/XmRec**. For example, the recording file for a console named:

```
Remote IP Load
```

would be:

```
$HOME/XmRec/R.RemoteIPLoad
```

Be aware of the following information when recording statistics:

1. When a file is created, a full description of the entire console is written as the first records of the file. This is true whether recording is started for the console as a whole or for only some instruments in the console.
2. If the file exists when you wish to start recording, you are asked whether you want to append to the file or re-create it. If you elect to append to the file, it is assumed that a console description already exists in the file.
3. Recording files are located in a subdirectory named **XmRec** in the user’s home directory. If this directory does not exist when recording is requested, **xmperf** attempts to create it.

### Recording Methods

When you select recording from a menu, you are presented with the following choices:

**Save Buffer**
: This option transfers the contents of the selected console’s or instrument’s history buffer to the recording file. The option is only available when recording is not already active for the console, because the values saved from the buffer would otherwise be out of synchronization with the ongoing recording’s values. The option is intended for situations where you detect an interesting pattern in a console and want to capture it for later analysis. When the recording of the buffer is completed, the recording file is closed.

**Begin Recording**
: This option starts writing data values to the recording file as they are received. Recording continues until you stop it or the console is closed.

**Save & Begin Recording**
: Combines the previous two options by first saving the buffer data to the file and then starting recording.

© Copyright IBM Corp. 1994, 2004

**End Recording**
Stops recording and closes the recording file if no other instrument in the console is still recording.

**Annotation** Allows a user to add annotations to a recording while the recording is taking place. Annotations are described in

## Active Recording Menu Items

Depending on whether a console or one of its instruments is currently recording, and depending on which recording menu you select, different items in the recording submenu are active. If you understand this, you can record information that you need while avoiding recording unwanted information. The status of the menu items is closely related to the difference between console recording and instrument recording, so look at that first.

If the Recording submenu is arrived at from a Start Console Recording menu, all menu items in the submenu are assumed to be concerned with the console as a whole. Thus, whether one, more, or all instruments in the console are currently being recorded, a selection of **End Recording** stops recording for all instruments. Similarly, no matter if one or more instruments are currently being recorded, a selection of **Begin Recording** from the submenu causes all instruments in the console to be recorded from this time on.

If the recording submenu is arrived at from a Start Instrument Recording menu, all menu items in the submenu are considered to apply to the currently selected instrument. Therefore, if the selected instrument is not currently being recorded, a selection of **Begin Recording** starts recording for the instrument. If the instrument is being recorded, no matter if the recording was started as a consequence of a full console recording being started, a selection of **End Recording** stops recording for the selected instrument. In neither case does the operation affect any other instrument in the console.

The **Save Buffer** submenu item is valid only if no recording is currently going on for any instrument in the console. This restriction helps you avoid mixing historic data with a “real-time” recording.

All the previous rules influence what submenu items are active at any point in time. The following table lists the possible combinations:

| Menu type | Selection | All instruments are recording | Selected instrument is recording | Selected instrument not recording | No instrument selected | No instrument is recording |
|---|---|---|---|---|---|---|
| Console Menu | Save Buffer | - | - | - | - | + |
| Console Menu | Begin Recording | - | + | + | + | + |
| Console Menu | Save & Begin Rec | - | - | - | - | + |
| Console Menu | End Recording | + | + | + | + | - |
| Console Menu | Annotate | + | + | + | + | - |
| | | | | | | |
| Instrument Menu | Save Buffer | - | - | - | N/A | + |
| Instrument Menu | Begin Recording | - | - | + | N/A | + |
| Instrument Menu | Save & Begin Rec | - | - | _ | N/A | + |
| Instrument Menu | End Recording | + | + | _ | N/A | - |

| Menu type | Selection | All instruments are recording | Selected instrument is recording | Selected instrument not recording | No instrument selected | No instrument is recording |
|---|---|---|---|---|---|---|
| Instrument Menu | Annotate | + | + | _ | N/A | - |

+ means option is available under these conditions.
- means option is not available under these conditions.

Recording Submenu, Active Items

To remind you that recording is in progress, a symbol illustrating a tape reel is shown in the lower right corner of all instruments with recording active (except state light instruments).

## Playback of Recordings

Playback is initiated from the File menu of the main window of **xmperf**. When you select the **Playback** menu item, you are presented with a list of files available for playback. The file list consists of all files in the **$HOME/XmRec** directory with a prefix of "R." You can use the filter part of the file selection box to look for other masks in whichever directory you want. Select a file to replay, and then select **OK** or double-click on the file name.

The selection box remains open after you select a file to replay. This allows you to select multiple files. Select **Cancel** to remove the selection box..

## Creation of Playback Consoles

Recording files may have been produced by **xmperf**, in which case they contain information about the appearance of the console from which they were recorded. Recording files created or modified by **3dmon**, **xmservd**, **azizo** or one of the recording support programs do not ly contain console information.

If a recording file does contain console information, this information is used to create the playback console in the image of the original console. This cannot be done for other types of recordings so **xmperf** uses default layouts in playback consoles for such recordings.

Though recording files may lack console information, they must contain information that groups statistics together in sets. Each set is treated by **xmperf** as an instrument. If any set contains more than 24 statistics, the set is broken into smaller sets. The playback console for a recording without console information is built by adding instruments for the sets as they are encountered in the recording file. Initially, **xmperf** attempts to allocate space for the instruments in the full width of the console. If too many instruments or values are present to allow this, the console is divided into multiple columns of instruments.

### Default Value Properties

When a recording file lacks console information, it still can contain information about the color or style of values. Similarly, it can include or lack information about the scales to use for plotting the values. **xmperf** uses any such information present and supplies default properties when they are not. The default properties for values are:

**Color** Selected from the list of colors provided through the **ValueColorX** resources. No tile is selected.

**Style** The secondary style is set to None, which means the value is plotted in the primary style of the instrument.

**Lower range**
The lower range (low scale) is set to zero.

**Upper range**
: The upper range (high scale) is set as if the **Maxiscale** menu selection (see “Playback Console Windows” on page 43) had been applied, except that the value is never set to less than 100.

**Threshold**
: The threshold is set to zero.

**Threshold type**
: The threshold type is set to ascending.

### Default Instrument Properties

Recording files without console information never carry instrument descriptions. Therefore, a set of standard instrument properties is assigned to the instruments used to play the recorded statistics back. The default properties are the same as are assigned to a console you create from scratch. They are described in “Instruments” on page 12.

## Using the Playback Console

When you select a valid recording file, **xmperf** reads any console configuration from the file and creates the console. If console information is not available in the recording file, a default console is constructed. A playback console looks different from other consoles because a row of buttons is displayed below the top border of the console window. Playback does not start until you select **Play**. The following figure shows a playback console for a recording created from the console shown in Figure 1 on page 9.

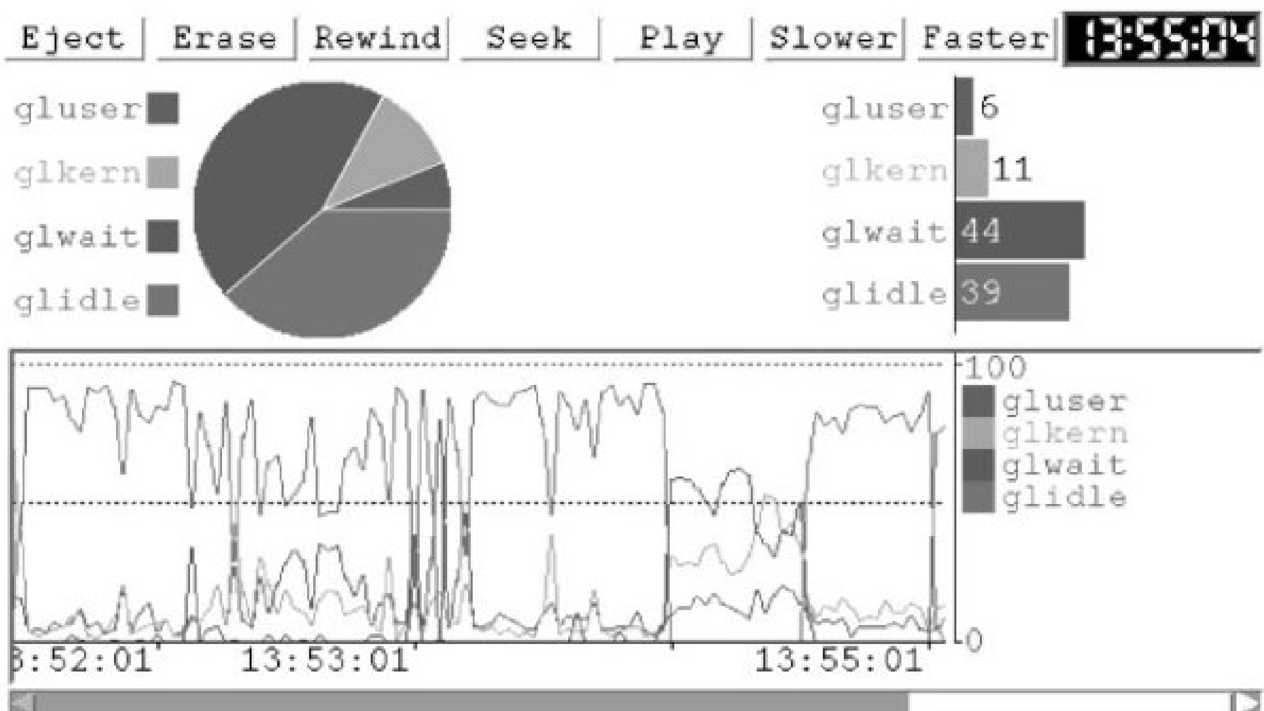



*Figure 3. Sample xmperf Playback Console. This console shows the control buttons at top (Eject, Erase, Rewind, Seek, Play, Slower, Faster) and the latest playback time. The upper-left pie chart displays the gluser, glkern, glwait and glidle metrics. The upper right bar chart displays the same metrics in bar format. The bottom display is a histrogram line chart of the same metrics with a scroll bar.*

The functions of the buttons at the top of the window are as follows:

**Eject**
: Immediately stops playback, closes the console, and closes the recording file. To restart playback, you must choose **Playback** from the File menu of the main window and reselect the recording file.

**Erase** Allows you to erase a recording file. When you select this button, a dialog box pops up. It warns you that you have selected the erase function and tells you the name of the file you are currently playing from. To erase the file and close the playback console, select **OK**. To avoid erasure of the file, select **Cancel**.

If some other program is using the recording file at the time you attempt to erase it or if you are not authorized to delete the file, you are informed of this and are prevented from erasing the file.

**Rewind**
Resets the console by clearing all instruments and rewinds the recording file to its start. Playback does not start until you select **Play**. The **Rewind** button is not active while you are playing back.

**Seek** Opens a dialog box that allows you to specify a time you want to seek for in the recording file. You can set the time by selecting on the **Hour** or **Minute** button. Each click advances the hour or minute by one. By holding the button down more than one second, you can advance the hour or minute counter fast. When the digital clock displays the time that you want to seek, select **Proceed**. This causes all instruments in the console to be cleared and the playback file to be searched for the time you specified.

In situations where a recording file spans over midnight so that the same time stamp exists more than once in the playback file, the seek proceeds from the current position in the playback file and wraps to the beginning if the time is not found. Because multiple data records may exist for any hour and minute combination, use the Play function to advance to the next minute before doing additional seeks on the same time, or seek for a time one minute earlier than the current playback time.

If you are playing back from a file while recording to the file is still in progress, the **Seek** function does not permit you to seek beyond the end time of the recording as it were when you first selected the file for playback.

The **Seek** button is not active while you are playing back.

**Play** Starts playing from the current position in the playback file. While playing, the button's text changes to **Stop** to indicate that playing can be stopped by clicking the button again. Immediately after opening the playback console, the current position is at the beginning of the recording file. The same is true after a rewind.

Initially, playing back is attempted at approximately the same speed at which the data was originally recorded. When the recording was created by other programs than **xmperf**, and especially if the file is produced by merging several files, **xmperf** might have difficulty determining this speed. This can cause the start of the playback to be delayed. You can change the speed by using the **Slower** and **Faster** buttons.

While playing back, neither the **Rewind** nor the **Seek** buttons are active.

**Slower**
Select this button to cut the playback speed to half of the current speed. Note that it may take a second for the new speed to become active.

**Faster** Select this button to double the playback speed. Note that it may take a second for the new speed to become active.

**00:00:00**
At the far right is a digital clock. It shows the time corresponding to the current position in the recording file or zeroes if at the beginning of the file. As playing back proceeds, the clock is updated.

## Recording File Inconsistencies

Recordings from instruments contain control blocks describing the instrument and console from which the recording was done. If a recording was created by a version of **xmperf**, which has a control block format different from the one of the version of the program used for playback, playback is impossible. When you attempt to play a recording back under such conditions, **xmperf** detects this and displays a dialog box. From the window you can choose to keep or delete the old recording file. You do not have the option of playing it back. If you choose to keep the file, you can convert it to **Performance Toolbox for AIX** format used by your version of **xmperf** with the **ptxconv** program.

Obviously, if you try to play back from a file that does not contain valid data, results are unpredictable. There may also be unanticipated side effects of special conditions for recording or playback, such as:

**Playing from saved buffers**
When the buffer of an instrument or console is saved, that buffer may not be full because the monitoring has not been going on for a long enough time. If you play such a recording back, the playback shows values of zeroes up to the point where real data values are available.

**Unsynchronized Instruments**
Playback from recordings of multiple data-supplier hosts in one console behaves just like the real console. Thus, time stamps are applied to each instrument (where applicable) as they are read from the data. This reflects the differences in time of day as set on the data-supplier hosts. However, be aware that these time differences influence the **Seek** function and the current position clock.

**Recordings from Instantiated Skeleton Consoles**
Each time a skeleton console is instantiated, the actual choices you make are likely to vary. This is no problem if you create recordings for each instantiated console, but if you append a recording to a previous one with the same name, things get complicated. The reason is that a recording contains the definition of the console only once (at the beginning of the recording). During playback, when you reach the position where a different instantiation was appended to a previous recording, it is assumed that the relative position of instruments and values is unchanged. It is unlikely to be unchanged, so you cannot trust the playback.

## Annotations

Annotations are special record types in recording files. They contain a several structured fields and a text of variable length. Because recording files are binary files, so are the annotation records. It requires special programs to add and delete annotation records. Several programs can be used to do this and one of those programs is **xmperf**. This section describes how to work with annotations. The description is centered around the way it is done in **xmperf** but applies to the other programs that support annotation. This includes **azizo**, **3dmon**, and **3dplay**.

## Annotation Types and Fields

In the current implementation, all annotations added interactively are timestamped annotations, meaning that they can be related to a specific point in time. The intention is to allow an annotation to describe a certain pattern of recorded metrics with reference to when it occurs in the recording. Annotations have the following structured fields:

**Timestamp**
The timestamp. If in playback mode, the timestamp refers to the time in the recording when annotation was selected. If in recording mode, the timestamp refers to the time of the recording when annotation is selected. The field is kept in internal format but displayed in text format by all programs using it.

**Status**
An annotation can be active or marked for deletion. This is shown by either `ACTIVE` or `DELETE` being displayed in the Annotation List window.

**Locale information**
Reserved for future use.

**Label** The abstract or label of the annotation. For example, the label of the first annotation in Annotation List window is the text `List of merged files`.

**Text** The actual text of the annotation. It is inserted automatically by a program.

## Annotating while Recording

Annotations can be added from **xmperf** and **3dmon** while recording is taking place. When you request annotation, a window opens that is empty except for the time stamp set to the time you requested annotation.

The label (abstract text) can then be entered, as well as the annotation text. The Add Annotation menu can be used to save the annotation to the recording file.

## Using Annotations

The most common way to view or add annotations is during playback of a recording file through **xmperf** or **3dmon**. It is also common during analysis with **azizo**. After selecting annotation, the Annotation List window opens and contains one line for each annotation in the recording file. Each line will contain the status, timestamp, and label.

From the pull-down File menu, **Add Note** or **Quit Annotation** can be selected. To add another annotation, use the pull-down File menu, and select **Add Note**. This will open the Adding an Annotation window, which was seen previously. **Quit Annotation** will exit annotation, and write all added notes to the recording file.

To view the annotation text, select the line that represents the annotation you want to see.

From the Annotation View window, notes can be deleted, undeleted, or quit.

**Delete Note** will mark the annotation as deleted. The notes will not actually be removed from the recording file. **Undelete Note** will mark the note as active. The notes remain in the recording file until **Write Current View** is selected from the playback menu. **Write Current View** will rewrite the recording file, omitting all notes marked deleted. This will reduce the size of the recording file.

# Chapter 5. The xmperf Command Menu Interface

The command menu interface of **xmperf** is entirely configurable. The intention with this interface is to:

- Provide a handy list of performance-related commands and utilities.
- Provide an easy way to define standard sets of options with which to execute any utility function.
- Provide a user-friendly way of selecting command line options and supplying command line values.

Virtually any command can be defined in the **xmperf** configuration file. A simple syntax is used to arrange the commands in multi-level menus and to define the specifics of each command and its options. The following sections describe how to define menus and commands to **xmperf**.

## Command Menus

Commands are defined as belonging to one of the following **xmperf** main window pull-down menu items:

**Analysis**
Intended to contain tools that can be used to analyze a specific application, environment or configuration. For example, this group would contain tools such as **tprof** (a tool to determine which part of a program most of the execution time is spent in) and **rmss** (a tool to simulate different real memory sizes).

**Controls**
Intended as a place to keep tools to change system parameters that influence performance. You can have tools to change the number of buffers, the number of **biod** daemons, and so forth.

**Utilities**
Intended for any remaining commands you'd like to assign to a menu.

Lines in the configuration file that define menus or commands must have an identifier as the first characters on each line. This identifier determines what main menu the definition goes to:

**analy** Analysis main menu

**ctrl** Controls main menu

**util** Utilities main menu

## Defining Menus

The command menus are cascading menus. This means that each item on any given menu represents either another level of menu or a single command. A menu tree for the main menu group Analysis might be represented like this sample menu structure:

```
Start New xmperf

Time Profiler
   Profiling w/source
   Minimal Profiling
Reduced Memory Simulator
   Memory Simulation
      Standard Simulation
      Special Simulation
   Set Reduced Memory
   Reset Memory
```

In the preceding example, menu lines are those followed by an indented line. The end-points of each indentation represent tools or commands that can be selected and executed. Therefore, both of the following lines are menu lines:

```
Time Profiler
Memory Simulation
```

© Copyright IBM Corp. 1994, 2004

Yet the following two lines represent executables:

```
Start New xmperf
Profiling w/source
```

Executables are defined as described in the following example, Defining the Sample Menu Structure.

**Note:** The menu structure defined previously is created with the following lines. (Lines enclosed in "<>" are place holders for lines to define executables; ignore this for now.)

```
   <lines to define: Start New xmperf>
analy.menubegin.Time Profiler
   <lines to define: Profiling w/source>
   <lines to define: Minimal Profiling>
analy.menuend.analy
analy.menubegin.Reduced Memory Simulator
analy.menubegin.Memory Simulation
   <lines to define: Standard Simulation>
   <lines to define: Special Simulation>
analy.menuend.analy
   <lines to define: Set Reduced Memory>
   <lines to define: Reset Memory>
analy.menuend.analy
```

The lines to define executables that are displayed between any pair of `menubegin` and `menuend`, are added to the menu named in the `menubegin` line. Whenever a new `menubegin` line is encountered, another menu level is begun.

A `menuend` line terminates the menu begun by the last `menubegin` in effect. Excess `menuend` lines are ignored.

Menu definition lines must follow the previously discussed format. The `menubegin` lines define the name of the menu. The name contains all characters following the `menubegin` keyword (and the period) up to the end-of-line character. The `menuend` lines must have a non-blank character following the `menuend` keyword (and a period). By convention, the name of the main menu group is used.

The first five (or six, in the case of the Analysis menu) characters on each menu line must be `analy`, `ctrl`, or `util` followed by a period.

## Executables

The following topics are discussed in this section:

- Defining Executables
- Defining Options for Executables.

## Defining Executables

An executable is defined by two line types. For both line types, the first five (or six) characters of each line must be one of `analy`, `ctrl`, or `util` followed by a period. The first five (or six) characters determine the menu to which the executable is added.

The characters following the first period and up to the next period (but not including the latter) identify the executable. The identification is used to group together all the lines that define a particular executable, and is also the text shown on the menu where the executable is added. After the period that terminates the identification must come one of the following:

**program:** A keyword to identify this line as the skeleton command line to be executed.

**$token:** The `$` character identifies this line as an options line. The characters between the `$` and the colon represent the name of a token and are used to match the defined option against a character string of the format `$token` in the skeleton command line.

As an example, define the executable represented by the following line in the Sample Menu Structure ( “Defining Menus” on page 61):

```
Set Reduced Memory
```

That could be done as follows:

```
analy.Set Reduced Memory.program: rmss $mem
analy.Set Reduced Memory.$mem: -c%r%n%16%Simulated Memory Size
(MB)
```

What these two lines say is that the command line to execute for this particular tool is:

```
rmss $mem
```

The token `$mem` should be substituted by whatever the user responds when the second line causes him or her to be prompted for `Simulated Memory Size`. The line with the `program:` keyword represents a skeleton command line. Before the command line is executed, **xmperf** attempts to replace all tokens in the skeleton line with the values the user selected. When attempting this, **xmperf** may end up in the following different situations:

**The user changed the token value.**
: In this case, the new value is taken and used to replace all occurrences of the token in the skeleton command line.

**The user erased the token value.**
: This can only happen when the token is defined as optional. When this happens, all occurrences of the token are removed from the skeleton command line.

**The user left the token value unchanged.**
: When this happens, and no default value is given (when the token is defined as optional), all occurrences of the token are removed from the skeleton command line. If a default value is given, this value is used to replace all occurrences of the token in the skeleton command line.

In all situations where a token value is replaced in (rather than removed from) the command line, the token value includes the command flag if defined in the option line. Where a token is removed from the skeleton command line, the flag is not inserted either.

Before the command line is submitted for execution, an & (ampersand) is added at the end of the line. This causes all commands to be executed in background.

## Defining Options for Executables

The general format of an option line is as follows, with the character % (percent sign) as a delimiter between fields:

```
menu.identifier.$token:
[flag]%{o|r}%{c|n|f|e}%[default]%description
```

**flag**
: Optional field. Used to specify the flag (or nothing) that must precede the command line argument when the token is replaced in the command line. If no flag value is given, none is included when the token substitution is done. For some commands, a blank must separate the flag value from the argument value. In such cases, the flag defined in the options line must have a trailing blank.

**{ o | r }**
: Required field. Specifies whether the option described by this line is required for proper execution of the command or not. A value of **o** denotes optional, while **r** means that the value is required. **xmperf** uses this field to determine whether it is acceptable for the user to erase a default value or leave an option value empty.

**{ c | n | f | e }**
: Required field. The field defines the option type and must be one of the following:

  **c** A character input line. This option line causes the user to be prompted to enter or change a character value such as a path name.

**n** A numerical input line. This option line causes the user to be prompted to enter or change a numerical value such as a count or size.

**f** A flag definition line. This option line either defines a single option (when it is not followed by extension lines) or a group of options (as defined by succeeding extension lines).

**e** A flag extension line. This line is ignored if it it does not match a previously encountered flag definition line. Multiple extension lines define a group of options. If the corresponding flag definition line is marked as optional, the group of options is treated as multiple choice, meaning that one, more, all, or none of the options can be selected. If the corresponding flag definition line is marked as required, the group of options is treated as single choice, meaning that one, and only one, of the options must be selected.

Some commands require a list of extension values to be separated by a non-blank character such as a comma. For such commands, the flag value of the extension line must be followed by that character on all extension lines. When the resulting command line is generated by **xmperf**, the last such character is removed.

**default** Optional field. The field is used differently, depending on the option type of the line:

**c or n** The contents of the default field are used to display an initial value in the user input field.

**f** If the line is not followed by extension lines, a value means that this option is selected by default. If no default value is specified, the option is not selected by default. By convention, if a default value is given, it is specified as the character "y". If the line is followed by extension lines the default value is ignored.

**e** For extension lines belonging to an optional flag definition line, each line with a default value specified is selected by default. For extension lines belonging to a required flag definition line, the first extension line with a default value specified is selected by default; default values on following lines are ignored.

**description** Required field. Specifies the prompt text associated with the defined option.

The following sections show a few examples to clarify how to define some more complex executables. This first example shows how you might define one particular invocation of the **svmon** analytical tool.

## Example svmon Definition

The **svmon** tool has many command line options. This example defines only a subset of options, namely the flags **-P** (output report selection based on processes) and **-i** (time between iterations and number of iterations). The syntax of the selected options is:

```
[-P{n|s|a}{u|p|g} [count]]
```

and:

```
[-i interval [count]]
```

This means that both flags are optional, as seen from the point of view of **svmon**, but elect to define them as required because you are defining one particular invocation of the tool. When the **-P** flag is specified, one argument of each of the groups **nsa** and **upg** must be selected, which is a good opportunity to show you a definition of a **single choice** options list. A count may or may not be given.

The **-i** flag must be followed by an interval between executions and, optionally, by a number of executions. Because the default number of executions is limitless, define this number as required.

The lines to define this executable could look like this:

```
analy.svmon sample.program: (svmon $1$2 $count $int $rep
>$ofile; \
                    sleep 10; aixterm -e view $ofile )
```

```
analy.svmon sample.$1:      -P%r%f%%Output selection
analy.svmon sample.$1:      n%r%e%y%Non-system segments only
analy.svmon sample.$1:      s%r%e%%System segments only
analy.svmon sample.$1:      a%r%e%%All segment types
analy.svmon sample.$2:      %r%f%%Process sort criteria
analy.svmon sample.$2:      u%r%e%y%Decreasing by pages in real memory
analy.svmon sample.$2:      p%r%e%%Decreasing by pages pinned
analy.svmon sample.$2:      g%r%e%%Decreasing by page space used
analy.svmon sample.$count:  %o%n%10%Number of process reports
analy.svmon sample.$int:    -i %r%n%5%Interval, number of seconds
analy.svmon sample.$rep:    %r%n%3%Number of iterations
analy.svmon sample.$ofile:  %r%c%./svmon.out%Filename for svmon output
```

## Defining an Execution of svmon

The `program:` line is a skeleton for the execution of a series of commands:

- The **svmon** command itself.
- The **sleep** command to induce a 10 second delay.
- The **aixterm** command to open a window and show the output from **svmon** using the **view** command.

The commands are separated by semicolons and the entire skeleton line is enclosed in parentheses, ready for execution under **ksh**. Notice that when the definition is split over two lines, you terminate the first line with a \ (backslash); this can be continued up to a maximum of 1024 characters. The skeleton command line specifies tokens to be substituted for values chosen by the user. Each token is positioned at the appropriate place for correct execution of the command.

One token is not part of the **svmon** command line. That's the token called **$ofile**, which is used to specify a file name where the output from **svmon** is to be written. Because **svmon** writes its output to `stdout`, the token is used to redirect the output to the file name given. Notice how this token is displayed twice in the skeleton command line: once to do the redirection; once to specify the input file to **view**.

The option lines are defined so that the skeleton command line would be converted to the following line if the user changes nothing:

```
(svmon -Pnu 10 -i 5 3 >./svmon.out; sleep 10; aixterm -e
view ./svmon.out)
```

## Example vmstat Definition

The command **vmstat** has several command line options. The syntax of the **vmstat** command line is:

**vmstat** [**-f**] [*Phys_volume* ...] *[interva*l [*count*]]

When **vmstat** executes, it sends its output to `stdout`. Because you want to be able to browse through the output, send it to a file and then use **view** to browse that file.

A challenge is the definition of **interval** and **count**, because the omission of the first would cause **vmstat** to use the second as **interval**. To overcome this, define both options as being required and give reasonable default values.

The definition looks as shown in the following example:

```
analy.vmstat sample.program: (/usr/ucb/vmstat $forks $vols
$int $iter\
                             > $ofile; aixterm -n vmstat_out\
                             -name vmstat_out -e view $ofile )
analy.vmstat sample.$forks:  -f%o%f%%Report number of forks since boot
analy.vmstat sample.$vols:   %o%c%%List of physical volumes to analyze
analy.vmstat sample.$int:    %r%n%5%Interval between samples
analy.vmstat sample.$iter:   %r%n%20%Number of samples
analy.vmstat sample.$ofile:  %r%c%$HOME/vmstat.tmp%Output file name
```

### Defining an Execution of vmstat

With this definition, you can accept that the user keys in a list of physical volumes to analyze in response to the prompt for the token **$vols**. This requires you to remember or look up the names of physical volumes. To make things easier, you could code this into the definition. Assuming the physical volumes **hdisk0** through **hdisk2**, defining an enhanced execution of vmstat looks as shown in the following example.

```
analy.vmstat sample.program: (/usr/ucb/vmstat $forks $vols
$int $iter\
                            > $ofile; aixterm -n vmstat_out\
                            -name vmstat_out -e view $ofile )
analy.vmstat sample.$forks:  -f%o%f%%Report number of forks since boot
analy.vmstat sample.$vols:   %o%f%%Select physical volumes to analyze
analy.vmstat sample.$vols:   hdisk0 %o%e%y%Physical volume: hdisk0
analy.vmstat sample.$vols:   hdisk1 %o%e%y%Physical volume: hdisk1
analy.vmstat sample.$vols:   hdisk2 %o%e%y%Physical volume: hdisk2
analy.vmstat sample.$int:    %r%n%5%Interval between samples
analy.vmstat sample.$iter:   %r%n%20%Number of samples
analy.vmstat sample.$ofile:  %r%c%$HOME/vmstat.tmp%Output file name
```

### Defining an Enhanced Execution of vmstat

This is an example of how you specify a group of flag extensions to be presented in a *multiple choice* selection prompt. The advantage of using this technique rather than defining each disk as a separate flag is that you can use the flag definition line to specify a heading for this group of choices. Notice how each of the flag values represent data values rather than traditional flags and that each of the values has a trailing blank. Without the trailing blanks, all choices would be sent to **vmstat** as one concatenated character string.

## An Alternative vmstat Definition

When you use the technique described in “Example vmstat Definition” on page 65 to redirect output from a command running in background to a file and then use the **view** command to show it in an aixterm window, you don’t have any way of knowing that a popup window should open. Likewise, you might miss the fact that the commands run in the background and try to run a command multiple times not realizing that it’s already executing.

This is inherent in UNIX systems, but it becomes less apparent when the command is executed from a menu. Thus, the typical operating system response to submitting a background job is not seen by the user. The following example of a **vmstat** definitionis taken from the distributed configuration file. The example shows how a clever use of parentheses and double quotation marks can direct the output from the **vmstat** command to an aixterm window. You can use the window’s scroll bar to examine the output.

```
analy.vmstat Monitor.program: aixterm -geometry 85x20 -n
vmstat \
                              -name vmstat -sl 1000 -sb -fn Rom10\
                              -e ksh -c "(/usr/ucb/vmstat \
                              $(cd /dev ; /bin/ls cd* hdisk*) \
                              $int $iter; read)" &
analy.vmstat Monitor.$int:    %r%n%5%Interval between samples
analy.vmstat Monitor.$iter:   %o%n%20%Number of samples
```

### An Even More Enhanced Definition of vmstat

In the preceding example, the execution of **vmstat** starts is shown by opening an aixterm window as described by the first two lines. The window has the following characteristics:

- The size is 85x20 characters
- Title bar and icon name are set to `vmstat`
- History of up to 1000 lines is saved
- Scroll bar is present
- Rom10 font is used

The third line specifies to execute the program **ksh** (`-e ksh`) and that the shell must run a command which follows the **-c** flag. The command to run is actually a series of commands, enclosed in double quotation marks as well as parentheses. The sequence of commands is as follows:

```
(/usr/ucb/vmstat $(cd /dev ; /bin/ls cd* hdisk*) $int $iter;
read)
```

The previous sequence means that the command `/usr/ucb/vmstat` is to be executed. The arguments passed to the command are the disk names produced by the command sequence:

```
cd /dev; /bin/ls cd* hdisk*
```

This command sequence is followed by the command line arguments represented by the tokens `$int $iter`. After this command is executed, issue a **read**, which terminates the **aixterm** session when the user presses the Enter key from the window.

## Process Controls

Two of the three main menu items that are used to define command menus have a fixed menu item. The main menu items are Controls and Utilities. This section describes the fixed menu item in the Controls pull-down. This fixed menu item is called **Process Controls**. The purpose of the **Process Controls** menu item is twofold:

- To provide a fast and comprehensive overview over all running processes.
- To make it easier to execute commands that take a list of process IDs as argument.

The default sort criterion for the process list is defined to place the largest consumers of CPU resources (hot processes) at the top of the list.

## Process Overview

When you select **Process Controls**, you immediately see a list of all running processes in your system at the time you made the selection. The list shows the most interesting details about the processes, and initially is sorted in descending order after CPU percentage used by the process. An example of a local process list is shown in the following figure.

The fields in the list are, from left to right:

**Process ID (PID)**
The process ID of the process.

**Command Name**
The command executing.

**PRI**
The priority of the process.

**Login User-ID**
The user ID used to login to the session from where the process is started.

**Effect User-ID**
The effective user ID for the process, as changed by **setuid** or **su** if applicable.

**Data Res (4 KB Pages)**
The number of 4KB (kilobytes) pages of real memory currently used for data segment by the process.

**Text Res (4 KB Pages)**
The number of 4KB (kilobytes) pages of real memory currently used for text segment by the process.

**Page Space (4 KB Pages)**
The number of 4KB (kilobytes) currently allocated for paging space on an external disk for this process.

**Latest CPU Perct**
When you create the process list from the Controls menu, this field shows the CPU usage of the process over its life time. The same is true

whenever a new process shows up on the process list after a refresh. In all other cases, this field shows the average CPU usage since the last refresh of the process list.

By default, this field is used to sort the list so you can easily pinpoint processes that are large consumers of CPU resources.

**Process Total (CPU Seconds)**
The sum of kernel-CPU and user-CPU time (in seconds) used by the process.

**Accum Total (CPU Seconds)** The sum of kernel-CPU and user-CPU time (in seconds) used by the process and all its children processes.

**Page-Faults, I/O** The number of page faults taken by the process to handle file I/O since the process was started.

**Page-Faults, other** The number of page faults taken by the process for other than file I/O since the process was started.

**Parent Relationship** A number of # characters to illustrate the nesting of processes. The more # (number signs), the deeper the process is in the process hierarchy.

## Process Overview Menu

When the process overview list is displayed, a menu bar is available to control the list. The following menu items are available:

File This menu item allows you to close the list (which simply removes the list from the display) or to refresh the list by reading the current process information from the system. When you select **Refresh**, the displayed list is removed and another is created. However, the new list is sorted the same way the old one was. Note that he process list is not updated by **xmperf** automatically. It is your responsibility to use the **Refresh** menu item to have the list updated as needed.

Sort The process list can be sorted on any of the fields shown. Sorting is in ascending order if one of the leftmost five fields is used for sorting, otherwise in descending order. Each time a list is sorted, **xmperf** remembers the sort criteria you chose. Whenever the list is refreshed, and when it is removed and later created again, the latest sort criteria is used to sort the list.

One special sort criteria is called “Parent Relationship” which sorts the processes so that a process is listed before its children. This generates a hierarchical display of parent-child process relationships at the far right of the list.

Execute This menu selection is another custom-built menu. Items are created as described in “Command Menus” on page 61 and “Executables” on page 62. This means that you can build a menu hierarchy and that individual commands can be located anywhere in that hierarchy.

Two things distinguish the Process Execute menu from other tools menus: the identifier for lines to define the line is *proc*, and a new token type is introduced. This is explained in “Process Token” on page 69.

The Execute menu selection is only active if the process overview window is generated by a selection of the **Process Controls** item in the Controls pull-down menu of the main window. If the process overview window is generated while instantiating a skeleton console, the **Done** menu item is active instead of **Execute**.

Done This menu selection is associated with the use of the process overview to instantiate skeleton instruments that have the process ID as a wildcard. The pull-down menu you get when you select **Done** has three menu items. If you select the **Cancel** menu item, the box goes away and no instrument is created. If you select the **Reselect** menu item, the process overview list is refreshed. If you select the **Accept Selection** menu item, the skeleton instrument is instantiated with the processes you selected.

Help Displays any help text supplied in the simple help file and identified by the name *Process Controls*.

## Process Token

Executables that are defined as belonging to the process execution menu must be defined so a process token is present in the skeleton command line. The process token is the # (number sign) character. During token substitution before the command is executed, the process token is replaced by a list of process IDs as selected from the process list by you.

If you try to execute a command from the Process Execution menu and no process ID is selected, the execution is not allowed. Selection can be done as a single-selection of one process by clicking on a line, or as a multiple-selection by dragging the mouse pointer with left mouse button pressed down. Extensions or deletions from selections can be done by holding the Ctrl key down on the keyboard while selecting.

Before the command is executed, you are presented with a the list of processes you selected. This way you can verify that the selection is correct.

## Example Definition for renice Command

To illustrate how you use the process token, the following example of using the process token shows how you could define the **renice** command in the Process Execution menu:

```
proc.Increase priority.program:  renice -n -$pri -p #
proc.Increase priority.$pri:     %r%n%1%Priority increase
proc.Decrease priority.program:  renice -n $pri -p #
proc.Decrease priority.$pri:     %r%n%1%Priority decrease
```

# Chapter 6. 3D Monitor

This chapter provides information about the **3dmon** program.

A system administrator is often responsible for monitoring the performance of several hosts in a network. To cope with this job, it is desirable to have a monitor that allows the monitoring of selected key statistics on multiple hosts. One such monitor is available in the program **3dmon**.

The **3dmon** program is an X Window System based program that displays statistics in a 3-dimensional graph where each of the two sides may have up to 24 statistics for a maximum of 576 statistics plotted in a single graph.

## Overview of the 3dmon Program

The graph layout of **3dmon** is that of a chessboard except that each side may have from 1 to 24 fields. The right side always represents a context in the statistics hierarchy. This context may be any context that can have more than one instance of its subcontexts. The currently valid contexts are:

```
CPU
Disk
FS
FS/*
IP/NetIF
LAN
Mem/Kmem
PagSp
Proc
RTime/ARM
RTime/ARM/*
RTime/LAN
hosts
```

**Note:** Not all wildcards are available on systems other than IBM RS/6000 systems.

## User Interface

For each context you want to use, you must supply a list of statistics in the **3dmon** configuration file. The list of statistics you specify is used to define a *set of statistics* (statset). In the configuration file, each set is defined by a wildcard stanza which names the configuration set. When you start **3dmon**, you specify which of the defined configuration sets should be used and as **3dmon** starts, it displays a list of all instances, which match the wildcard specified in the configuration set.

You must then pick the instances you want to monitor. The number of instances you select determines the number of fields on the right side of the “chessboard”. For certain configuration sets, you must first select from a list of hosts to monitor, then from a list of context instances on the selected hosts.

To select from the lists displayed by **3dmon**, move the mouse pointer to the first instance you want, then press the left mouse button and move the mouse while holding the button down. When all instances you want are selected, release the mouse button. If you want to select instances that are not adjacent in the list, press and hold the Ctrl key on the keyboard while you select. When all instances are selected, release the key. Using the left mouse button (after all selections are complete), select the button at the top of the window. This will create the 3dmon monitor window. An example of such a window is shown in the following figure.

© Copyright IBM Corp. 1994, 2004

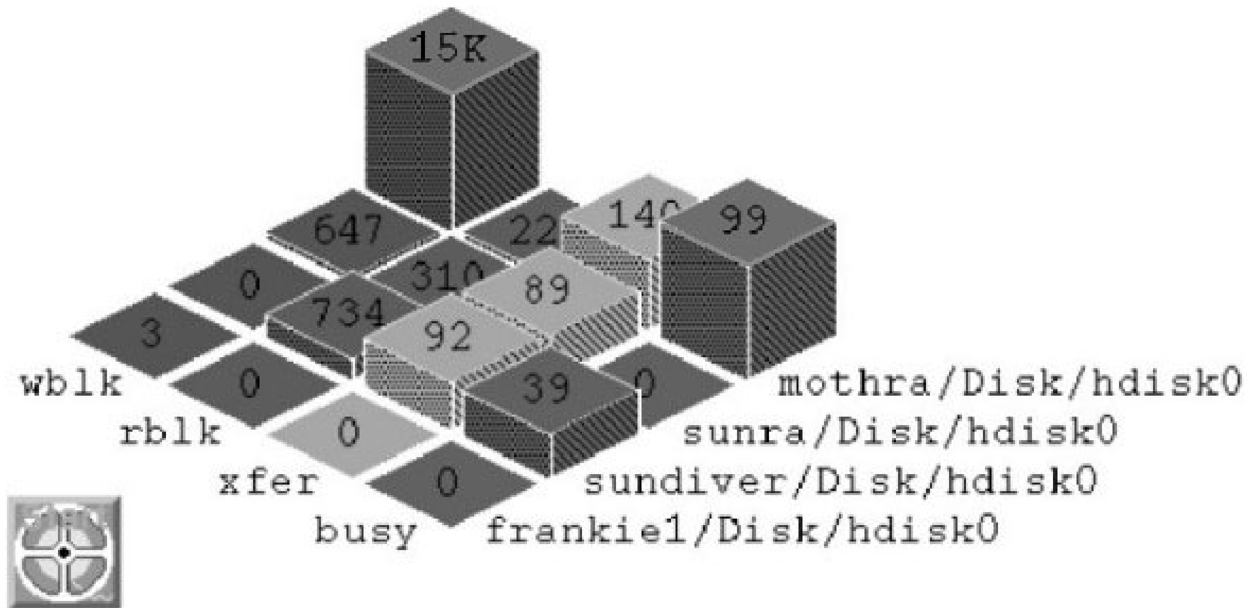



*Figure 4. 3dmon Graph. This graph displays a three-dimensional grid of metric names (y-axis) and associated hostnames (x-axis). The intersecting vertical "z" axis for each element of the grid represent the metrics value, which is shown as a vertical bar labeled with a numeric value. The activity off each system can be determined by comparing the relative displacements of the bars.*

The left side of the grid lists the statistics you chose to include in the configuration set. Each must be specified in the configuration file with its full value path name without the part up to and including the primary wildcard. This is explained in detail in "Customizing the 3dmon Program" on page 76.

The third dimension is represented by the actual statistics values as received from the data supplier daemons. The values are plotted as rectangular pillars placed on the fields of the chessboard, each filling its field except for a user modifiable spacing between the pillars. The actual value is displayed at the top of each pillar.

The **3dmon** graph shows the incoming observations of the values as they are received depending on the type of statistic selected. Statistics can be of type **SiCounter** or of type **SiQuantity**:

**SiCounter** Value is incremented continuously. The graph shows the change (delta) in the value between observations, divided by the elapsed time, representing a rate per second.

**SiQuantity** Value represents a level, such as memory used or available disk space. The actual observation value is shown by the graph.

A user-modifiable color selection is in effect so that each of the statistics represented by the left side of the grid is drawn in a different color.

## Pull-down Menus

The File menu of the 3dmon graph window contains the following valid selections:

**Playback** This option invokes 3dplay. See Chapter 7, “3D Playback,” on page 83 for details.

**Exit** Stops recording and exits program.

The Recording pull-down menu contains the following valid selections:

**Begin Recording**
This option starts writing data values to the recording files as they are received. Recording continues until you stop it or the console is closed.

**End Recording**
Stops recording and closes the recording file if no other instrument in the console is still recording.

**Annotation** Allows you to add an annotation to the currently active eller dormant recording file. When you select this item, proceed as described in “Annotating while Recording” on page 59.

## Autoscaling

To determine the height of a pillar, **3dmon** must select a scale. This scale states how many units of the statistic to be drawn correspond to one pixel on the display. The scale is derived from the expected maximum value for the statistic. However, it is difficult to set a realistic scaling for many statistics. Because of that, **3dmon** attempts to adjust the scale when the drawn pillars would otherwise disappear off the top of the window. The autoscaling is done as the following describes.

Prior to a pillar being drawn, if the **3dmon** sees that the pillar would be more than 10 percent taller than the expected maximum height, an autoscaling function is invoked. The autoscaling function slowly adjusts the scale of a statistic until the scale corresponds to actual measurements. The scale is adjusted by approximately 50 percent each time autoscaling is invoked. Also, autoscaling is always done for all instances of the statistic as represented by a row on the chessboard.

## Resynchronizing with Multiple Hosts

When **3dmon** uses the wildcard value *hosts*, the user selects the hosts to monitor from the initial selection list. If the **xmservd** daemon on one or more of those hosts dies while **3dmon** runs, the program periodically attempts to resynchronize with the hosts. Note that with the Performance Toolbox Local feature, only the local host is available for selection.

Resynchronizing is attempted every 30 seconds if more than 30 seconds have expired since the last data feed was received from a host. The interval between checking if resynchronizing is required can be changed with the command line argument **-t**.

## Viewing Obscured Statistics

As **3dmon** displays the statistics, high values for statistics in the front part of the grid can obscure the drawing of statistics behind them. At any time, you can move a row or column to the front by clicking on the name associated with the row or column. When you select a name on the left side of the grid, the colors of statistics are preserved as the rows of statistics are rearranged.

Another way to see obscured statistics is to reduce the height of the **3dmon** window. Because the height of the pillars is calculated from the ratio of window height to window width, decreasing the height reduces the height until only the base of the pillar is drawn.

## How to Record with 3dmon

In the lower left corner of the **3dmon** window, you see a small icon resembling a tape reel. Initially, **Start** shown in green appears on the icon. When you select the icon with the left mouse button, recording of the statistics received by 3dmon begins and the text changes to **Stop** shown in red. The recording facility is described in “Recording from 3dmon” on page 80.

You can also start and stop recording from the pull-down menu.

## Path Name Display

The right side of the grid shows the path names of the selections you made from the selection list up to the part that is displayed at the left side of the grid. If all the names at the right side begin with the same string, that string is removed from the names and shown in the upper left corner of the window. This reduces path name lengths and leaves more space to show the graph.

# The 3dmon Command Line

To avoid clashes with X Window System command line options, never leave a blank between a command line option and its argument. For example, do not specify

```
3dmon -i 1 -p 75 -n
```

Instead, use:

```
3dmon -i1 -p75 -n
```

The **3dmon** program takes the following command line arguments, all of which are optional:

**3dmon** [**-vng** ] [**-f** *config_file*] [**-i** *seconds_interval*] [**-h** *hostname*] [**-w** *weight_percent*] [**-s** *spacing*] [**-p** *filter_percent*] [**-c** *config*] [**-a** "*wildcard_match_list*"] [**-t***resync_timeout*] [**-d** *invitation_delay*] [**-l** *left_side_tile*] [**-r** *right_side_tile*] [**-m** *top_tile*]

**-v** Verbose. Causes the program to display warning messages about potential errors in the configuration file to stderr. Also causes **3dmon** to print a line for each statset created and for each statistic added to the statset, including the results of resynchronizing.

**-n** Only has an effect if a filter percentage is specified with the **-p** argument. When specified, draws only a simple outline of the grid rectangles for statistics with values that are filtered out. If not specified, a full rectangle is outlined and the numerical value is displayed in the rectangle.

**-g** Usually, **3dmon** will attempt to resynchronize for each statset it doesn't receive data-feeds for for resync-timeout seconds. If more than half of the statsets for any host are found to not supply data-feeds, resynchronizing is attempted for all the statsets of that host. By specifying the **-g** option, you can force resynchronization of all the statsets of a host if any one of them becomes inactive.

**-f** Allows you to specify a configuration file name other than the default. If not specified, **3dmon** looks for the file **$HOME/3dmon.cf**. If that file does not exist the file is searched for as described in Appendix B, "Performance Toolbox for AIX Files," on page 271.

**-i** Sampling interval. If specified, this argument is taken as the number of seconds between sampling of the statistics. If omitted, the sampling interval is 5 seconds. You can specify from 1 to 60 seconds sampling interval.

**-h** Used to specify which host to monitor. This argument is ignored if the specified wildcard is "hosts." If omitted, the local host is assumed. With the Performance Toolbox Local feature, this flag always uses the local host name.

**-w** Modifies the default weight percentage used to calculate a weighted average of statistics values before plotting them. The default value for the weight is 50%, meaning that the value plotted for statistics is composed of 50 percent of the previously plotted value for the same statistic and 50 percent of the latest observation. The percentage specified is taken as the percentage of the previous value to use. For example, if you specify 40 with this argument the value plotted is:

```
.4 * previous + (1 - .4) * latest
```

Weight can be specified as any percentage from 0 to 100.

**-s** Spacing (in pixels) between the pillars representing statistics. The default space is 4 pixels. You can specify from 0 to 20 pixels.

**-p** Filtering percentage, **-p**. If specified, only statistics with current values of at least **-p** percent of the expected maximum value for the statistic are drawn. The idea is to allow you to specify monitoring "by exception" so statistics that are approaching a limit stand out while others are not drawn. Filtering can be specified as any percentage from 0 to 100. Default is 0%.

**-c** Configuration set. When specified, overrides the default configuration set and causes **3dmon** to configure its graph using the named configuration set. The argument specified after the **-c** must match one of the **wildcard** stanzas in the configuration file. If this argument is omitted, the configuration set used is the first one defined in the configuration file.

**-a** Wildcard match list. When specified, is assumed to be a list of host names. If the primary wildcard in the selected configuration set is **hosts**, then the list to display host names is suppressed as **3dmon** automatically selects the supplied hosts from the list of active remote hosts. Depending on the configuration set definition, **3dmon** then either goes directly on with displaying the monitoring screen or, when additional wildcards are present, displays the secondary selection list.

**Note:** With the Performance Toolbox Local feature, this flag always uses the local host name.

The list of host names must be enclosed in double quotation marks if it contains more than one host name. Individual host names must be separated by white space or commas.

The primary purpose of this option is to allow the invocation of **3dmon** from other programs. For example, you could customize NetView to invoke **3dmon** with a list of host names, corresponding to hosts selected in a NetView window.

**-t** Resynchronizing timeout. When specified, overrides the default time between checks for whether resynchronizing is required. The default is 30 seconds; any specified timeout value must be at least 30 seconds.

**-d** Invitation delay. Allows you to control the time **3dmon** waits for remote hosts to respond to an invitation. The value must be given in seconds and defaults to 10 seconds. Use this flag if the default value results in the list of hosts being incomplete when you want to monitor remote hosts.

**-l** (Lowercase L). Specifies the number of the tile to use when painting the left side of the pillars. Specify a value in the range 0 to 8. The values correspond to the tile names:

- 0: foreground (100% foreground)
- 1: 75_foreground (75% foreground)
- 2: 50_foreground (50% foreground)
- 3: 25_foreground (25% foreground)
- 4: background (100% background)
- 5: vertical
- 6: horizontal
- 7: slant_right
- 8: slant_left

The default tile number for the left side is 1 (75_foreground).

**-r** Specifies the number of the tile to use when painting the right side of the pillars. Specify a value in the range 0 to 8. The values correspond to the tile names specified previously for option **-l**. The default tile number for the right side is 8 (slant_left).

**-m** Specifies the number of the tile to use when painting the top of the pillars. Specify a value in the range 0 to 8. The values correspond to the tile names specified previously for option **-l**. The default tile number for the top is 0 (foreground).

## Hardware Dependencies

On some graphics adapters in certain configurations, the **3dmon** program might not give you proper tiling. If you notice this, use the following command line arguments to suppress tiling:

```
3dmon -l0 -r0 -m0
```

Use the flags shown in addition to any other flags you might require. You can substitute the digit `4` for any of the zeroes shown previously. The digit `0` means to paint the pillar in the foreground color; the digit `4` means to paint it in the background color.

## Exiting 3dmon

To exit the **3dmon** program, select **Exit** from the pull-down menu or close the program's window with the window manager. In case of the **mwm** window manager, click in the upper left corner of the window frame, then select the **Close** option from the menu.

# Customizing the 3dmon Program

This section includes the following topics:

- 3dmon Configuration File
- Single Wildcard Configuration Sets
- Dual Wildcard Configuration Sets
- Rsi.hosts File
- The 3dmon X Resources.

## 3dmon Configuration File

The **3dmon** program requires a configuration file to describe the set of statistics (statsets) to display. A configuration file can be specified with the **-f** command line argument. If it is not, **3dmon** first looks for the file **3dmon.cf** in your home directory. If no such file is found, the file is searched for as described in Appendix B, "Performance Toolbox for AIX Files," on page 271.

The configuration file may have comment lines beginning with the character # (number sign). It can contain multiple configuration sets, each of which must begin with a *wildcard* stanza. The wildcard stanza must have one or two arguments. The last argument must be a valid context name. Configuration sets can be constructed as single wildcard sets or dual wildcard sets. Single wildcard sets present only one selection list to the user, while dual wildcard sets require the user to select from two lists before **3dmon** begins to display statistics.

If the command line argument **-c** is not used to override the default, the first configuration set in the configuration file is chosen by **3dmon**. Otherwise the set named by the **-c** argument is chosen. If no set in the configuration file matches the **-c** argument, **3dmon** terminates with an error message.

## Single Wildcard Configuration Sets

The following is an example of a definition of a configuration set where only one argument is provided on the **wildcard** stanza. The argument is **hosts**, and is used both to identify the configuration set and to specify the wildcard context:

```
wildcard: hosts
```

If two arguments are supplied with the **wildcard** stanza, the first identifies the configuration set. This name is used to match any name passed with the command line argument **-c**. The second argument must be the name of the wildcard context.

The remaining lines of a set must each specify a statistic, which is valid for the given wildcard. A set is terminated when another **wildcard** stanza is met or at end of file. The following example shows statistics defined for the wildcard **hosts**:

```
wildcard:        hosts     # remote hosts
Mem/Virt/steal
PagSp/pgspgout
PagSp/pgspgin
Proc/swpque
Proc/runque
PagSp/%totalused
Syscall/total
SysIO/writech
SysIO/readch
```

Notice how the full path name is specified except for the wildcard part. The resulting path name for the instance (host name) **birte** for the first statistic would be **hosts/birte/Mem/Virt/steal**.

If you want more than one configuration set that uses the **hosts** wildcard, enter two arguments on the **wildcard** line:

```
wildcard:      nodecpu    hosts
CPU/cpu0/user
CPU/cpu0/kern
CPU/cpu0/wait
CPU/cpu0/idle
```

The configuration set shown previously would be activated by the command line:

```
3dmon -cnodecpu
```

For process statistics, the statistic path name includes the name of the process context, which is constructed from the process ID, a ~ (tilde), and the name of the executing program. To reach a specific process, you can add a line that specifies either the process ID followed by the ~ (tilde), or the name of the executing program. The following example shows how to specify a statistic for the *wait* pseudo process, which (on the operating system UPs) always has a process ID of 516. Both lines point to the same statistic.

```
Proc/516~/usercpu
Proc/wait/usercpu
```

If you specify a name of a program currently executing in more than one process, only the first one encountered is found.

The distributed sample configuration file for **3dmon** defines two single wildcard configuration sets for the **hosts** wildcard. One (the default) is called **hosts**. The other is called **24** because it shows 24 statistics for each host selected.

To monitor selected statistics for processes, the configuration file might contain:

```
wildcard: Proc
usercpu
kerncpu
workmem
pagsp
```

The configuration set shown previously would cause **3dmon** to present you with a list of all processes in the host.

Two contexts are special because they can exist in multiple instances and have subcontexts that can also exist in multiple instances. One of these contexts describes file systems and is named **FS**. To allow **3dmon** to find all instances in both context levels, you can begin a statistics line with the wildcard character * (asterisk). This is a use of dual wildcards is described here because it, unlike other dual

wildcard configuration sets, presents you with only one selection list. The use of the asterisk in a path name is shown in the supplied configuration set that follows:

```
wildcard: FS
*/%totfree
*/size
ppsize
free
```

The statistics **%totfree** and **size** exist for every logical volume, while the remaining two exist for every volume group. In a system with two volume groups and a total of five logical volumes, this would yield five columns (right side of graph) and four rows (left side of graph). For example, the right side might show:

```
rootvg/hd4
rootvg/hd9var
rootvg/hd3
rootvg/hd1
newvg/hd10
```

The left side would show the four statistic names. Note that because the **ppsize** and **free** statistics are at the volume group level, they are identical across all four columns that are derived from **rootvg**.

The supplied configuration file contains the following single wildcard configuration sets:

```
wildcard: hosts          # remote hosts
wildcard: 24 hosts       # remote hosts, large set
wildcard: Disk           # local disks
wildcard: Kmem  Mem/Kmem  # local kernel memory
wildcard: Proc           # local processes
wildcard: LAN            # local lan adapters
wildcard: FS             # local file systems
wildcard: IP/NetIF       # local IP interfaces
wildcard: CPU            # local processors
wildcard: RTime/ARM      # local application response time
wildcard: lanresp hosts  # response time between multiple hosts
```

The configuration set *"lanresp"* for response time between multiple hosts is special because it will have identical labels on the left and right side of the grid. It uses the top context **RTime/LAN**, which always creates this type of graph and thus allows only a single metric to be specified. This is described in more detail in "Monitoring IP Response Time from 3dmon" on page 194.

## Dual Wildcard Configuration Sets

If a configuration set begins with a **wildcard** stanza that defines the wildcard as **hosts**, then the path names of the statistics belonging to that set may contain one or more asterisks in place of contexts that may exist in more than one instance. This is shown in the supplied configuration set called **proc**:

```
wildcard: proc hosts # processes on remote hosts
Proc/*/usercpu
Proc/*/kerncpu
Proc/*/workmem
Proc/*/codemem
Proc/*/pagsp
Proc/*/majflt
Proc/*/minflt
```

When you invoke **3dmon** to use this configuration set, you are presented first with a list of all matches of the primary wildcard, which is the list of hosts that responded to invitation. After you select the hosts you want to monitor, you are shown a list of all processes on those hosts. To proceed, select the processes you want to monitor. If you select more than **3dmon** can show, excess instances are ignored.

If you use the command line argument **-a** to specify a list of host names, the first selection list is not shown. Rather, all responding hosts that match a name in the **-a** list are automatically selected.

Another example of a dual wildcard configuration set is as follows:

```
wildcard: fs hosts # remote file systems
FS/*/*/%totfree LV
FS/*/*/size     LV
FS/*/ppsize     VG
FS/*/free       VG
FS/iget         FS
```

This time the statistics are at different levels so that the first two are logical volume statistics, the next two are volume group statistics, and the last one is a statistic directly under the **FS** context. To show this in the final graph, a suffix is specified after the name of each statistic. The first word of such a suffix is suffixed to the statistic name on the graph. Its only purpose is to show you the levels of the statistics in the graph.

The third example of a dual wildcard configuration set is the following set to create a 3dmon graph to monitor application response time and activity:

```
wildcard:  armresp hosts # application response time
RTime/ARM/*/*/resptime
RTime/ARM/*/*/respavg
RTime/ARM/*/*/respmax
RTime/ARM/*/*/respmin
RTime/ARM/*/*/good
RTime/ARM/*/*/aborted
RTime/ARM/*/*/failed
```

When one or more lines have the wildcard character * (asterisk), all lines in that configuration set must either be without wildcards or match the same pattern of wildcards. It would be not be valid to specify the following two metrics under the same wildcard:

```
FS/*/*/%totfree
Proc/*/usercpu
```

**3dmon** warns about such errors in the configuration file and ignores the offending lines.

The supplied configuration file contains the following dual wildcard configuration sets:

```
wildcard: disk hosts     # disks on remote hosts
wildcard: kmem hosts     # kernel memory on remote hosts
wildcard: proc hosts     # processes on remote hosts
wildcard: lan hosts      # LAN adapters on remote hosts
wildcard: fs hosts       # file systems on remote hosts
wildcard: ip hosts       # IP interfaces on remote hosts
wildcard: cpu hosts      # Processors on remote hosts
wildcard: armresp hosts  # application response time
```

## Rsi.hosts File

The **3dmon** program uses the RSi application programming interface throughout. This means that to locate potential data supplier hosts it uses the **RsiInvite** function call. This function relies on the file **$HOME/Rsi.hosts** to specify the rules for broadcasting **are_you_there** messages. (See Appendix B, “Performance Toolbox for AIX Files,” on page 271 for alternative locations of the **Rsi.hosts** file).

If the file does not exist, broadcasting is done only to hosts on the same subnet as defined for the network adapters in the host where **3dmon** runs. For details about the **$HOME/Rsi.hosts** file, refer to “How Data-Suppliers are Identified” on page 24.

## The 3dmon X Resources

The X Window System resource file for **3dmon** defines resources you can use to enhance the appearance of **3dmon** and is installed as **/usr/lib/X11/app-defaults/3Dmon**.

X Window System Resources for 3dmon (below) lists the defined resources for **3dmon**.

The sample file first defines the font to use. The resource name to define the font for **3dmon** is **GraphFont**. If you do not define this resource, **3dmon** tries to get a font name from the following resources:

- **graphfont**
- **FontList**
- **fontList**
- **Font**
- **font**

If none are defined, a suitable fixed-pitch font is used.

The next two lines define the foreground and background colors of the graph area. Finally, 24 lines define the default colors for the up to 24 statistics that can be plotted. In the following example, X Window System Resources for **3dmon**, the colors are the default colors. Only change the **ValueColorxx** resources if you do not like the defaults.

```
  #
  #    3dmon  options
  #
  *GraphFont:
-ibm-block-medium-r-normal--15-100-100-100-c-70-iso8859-1
  *DrawArea.background:  black
  *DrawArea.foreground:  white
  *ValueColor1:   ForestGreen
  *ValueColor2:   goldenrod
  *ValueColor3:   red
  *ValueColor4:   MediumVioletRed
  *ValueColor5:   LightSteelBlue
  *ValueColor6:   SlateBlue
  *ValueColor7:   green
  *ValueColor8:   yellow
  *ValueColor9:   BlueViolet
  *ValueColor10:  SkyBlue
  *ValueColor11:  pink
  *ValueColor12:  GreenYellow
  *ValueColor13:  SandyBrown
  *ValueColor14:  OrangeRed
  *ValueColor15:  plum
  *ValueColor16:  MediumTurquoise
  *ValueColor17:  LimeGreen
  *ValueColor18:  khaki
  *ValueColor19:  coral
  *ValueColor20:  magenta
  *ValueColor21:  turquoise
  *ValueColor22:  salmon
  *ValueColor23:  white
  *ValueColor24:  blue
```

X Window System Resources for **3dmon**

## Recording from 3dmon

The icon shown in the lower left corner of the **3dmon** window controls recording of the observations received by **3dmon**. Initially, Start that is shown in green is on this icon. Using the left mouse button, select the icon to start recording to a disk file of the statistics received by **3dmon**. Simultaneously, the text changes to **Stop** that is shown in red.

The first time you select the icon, the 3dmon Recording File Name window opens and prompts you to select a file name for the recording file. The window has a default file name constructed from the path name of the **xmperf** recording directory **$HOME/XmRec** followed by **R.3dmon.set**, where the part

following the last period is the name of the configuration set you are using. Change the name if required, and then select **OK** to start the recording. If the name you select exists, you will be prompted to overwrite the file or select a different name.

After recording has been started, it can be stopped and restarted by selecting the icon. Whenever recording is active, **Stop** (shown in red) appears on the icon. After you stop recording, the recording file is kept open by **3dmon** to allow quick resumption of the recording without going through the prompting for a recording file name. The recording file is closed when **3dmon** exits. You can also start and stop recording from the 3dmon Recording pull-down menu.

Recordings produced by **3dmon** defines sets of statistics (statsets) for each selection you did from the selection list (one for each name at the right side of the grid). The value records in the recording file correspond to sets of statistics (statsets). All **3dmon** recordings can be played back by **xmperf** but they lack a console definition and therefore use the default playback console format described in “Creation of Playback Consoles” on page 55.

**3dmon** recordings can also be played back by **3dplay**.

Recordings produced by **3dmon** can be analyzed by the **azizo** program and processed by the recording support programs.

# Chapter 7. 3D Playback

This chapter provides information about the **3dplay** program.

## Overview of the 3dplay Program

With **3dplay**, **3dmon** recordings can be played back in the same style as the one in which they was originally displayed. When **3dplay** is invoked with no argument, a file selection window opens and shows a list of files that match the filter **R.3dmon.***.

When a valid **3dmon** recording is provided as an input through the command line or through the file selection window, **3dplay** displays its playback window immediately.

The 3dplay application display format is similar to 3dmon, however, at the top of the window is a row of buttons, much like those of a video recorder. The buttons have the following functions:

**Eject**
Stops playback and exits **3dplay** program.

**Annotate**
Allows you to display a list of any existing recordings and to then show, modify, and delete any of those. It also allows the creation of new annotations. The technique and windows used are the same as described in the section “Using Annotations” on page 59.

**Erase**
Allows you to erase a recording file. When you select this button, a dialog box opens. It warns you that you have selected the erase function and tells you the name of the file you are currently playing from. To erase the file and close the playback console, select **OK**. To avoid erasure of the file, select **Cancel**.

If some other program is using the recording file at the time you attempt to erase it or if you are not authorized to delete the file, you are informed of this and are prevented from erasing the file.

**Rewind**
Resets the console by clearing all instruments and rewinds the recording file to its start. Playback does not start until you select **Play**. The **Rewind** button is inactive while you are playing back.

**Seek**
Opens a dialog box that allows you to specify a time you want to seek for in the recording file. You can set the time by clicking on the **Hour** or **Minute** button. Each click advances the hour or minute by one. By holding the button down more than one second, you can advance the hour or minute counter fast. When the digital clock shows the time that you want to seek, select **Proceed**. This clears all the instruments in the console and searches for the time you specified in the playback file.

When a recording file spans midnight so that identical time stamps exist multiple times in the playback file, the seek proceeds from the current position in the playback file. Then the seek wraps to the beginning of the playback file if the time is not found. Because multiple data records may exist for any hour and minute combination, use the Play function to advance to the next minute before doing additional seeks on the same time. Or you can seek for a time that is one minute prior to the current playback time.

If you are playing back from a file while recording to the file is still in progress, the Seek function does not permit you to seek beyond the end time of the recording as it were when you first selected the file for playback.

The **Seek** button is inactive while you are playing back.

**Play**
Starts playing from the current position in the playback file. While playing, the button’s text changes to **Stop** to indicate that playing can be stopped by selecting the button again. Immediately after opening the playback console, the current position is at the beginning of the recording file. The same is true after a rewind.

© Copyright IBM Corp. 1994, 2004 

Initially, playing back is attempted at approximately the same speed at which the data was originally recorded. When the recording was created by other programs than **xmperf**, and especially if the file is produced by merging several files, **xmperf** may have difficulty determining this speed. This may cause the start of the playback to be delayed. You can change the speed by using the **Slower** and **Faster** buttons.

While playing back, neither the **Rewind** nor the **Seek** buttons are active.

**Slower**
Select this button to cut the playback speed to half of the current speed. Note that it may take a second for the new speed to become active.

**Faster**
Select this button to double the playback speed. Note that it may take a second for the new speed to become active.

**00:00:00**
At the far right is a digital clock. It shows the time corresponding to the current position in the recording file or zeroes if at the beginning of the file. As playing back proceeds, the clock is updated.

## The 3dplay User Interface

You can invoke the **3dplay** from the following areas:

- Command line
- **xmperf** utilities
- **3dmon** playback.

### Command Line Invocation

The syntax to invoke **3dplay** from the command line is:

**3dplay** [*RecordingFile* ]

Where *RecordingFile* is the name of a recording file created by **3dmon**. If a non-3dmon recording file is provided as input, **3dplay** returns an error message and the recording file will not be played back.

### Invocation from xmperf

**3dplay** can be executed from the Utilities pull-down menu of the xmperf main window. The identification string to create the text shown on the Utilities pull-down menu is 3-D Playback. When you select **3dplay** from the Utilities pull-down menu, **3dplay** is invoked with no argument from a window.

### Invocation from 3dmon

The **3dplay** program can be invoked from the File pull-down menu of a **3dmon** graph window. When you select Playback, **3dplay** is invoked with no argument.

# Chapter 8. Monitoring Exceptions with exmon

The Exception Monitoring program, **exmon**, works with the **filtd** daemon described in Chapter 16, “Data Reduction and Alarms with filtd,” on page 183. The **filtd** daemon can generate exception packets based upon alarm conditions defined in the filtd configuration file. The **xmservd** daemon, on the host where **filtd** runs, forwards the exception packets from **filtd** to any remote Data Consumer program that subscribes to exception packets.

The **exmon** program is designed to provide a convenient facility for monitoring exceptions as they are detected on remote hosts. It does so by allowing its user to register subscriptions for exception packets from all or selected hosts in the network and to monitor the exception status in a graphical window. Like any other remote Data-Consumer program, **exmon** uses the Remote Statistics Interface (RSi) to communicate with remote hosts. Unlike traditional Data-Consumer programs, however, **exmon** consumes exception packets rather than data feed packets containing metrics values.

The **exmon** program is started from the command line. It does not accept command line arguments. When started, its first action is to broadcast an invitation to all **xmservd** daemons in the network, according to the file **$HOME/Rsi.hosts**. For details about this file, refer to “How Data-Suppliers are Identified” on page 24. From the responses to the invitation, a list of host names is displayed for the user to select the hosts of interest. After the user selects the hosts, the **exmon** main window is displayed.

**Note:** With the Local Performance Toolbox option, available with Version 2.2 or later, only the local host will be used. Any discussion of remote hosts apply only to the local host if the Local Performance Toolbox is installed.

## The exmon Main Window

At the top of the main window is a menu bar. Below the menu bar is a graphical window that contains the monitoring window.

## The exmon Monitoring Window

The layout of the **exmon** monitoring window is that of a matrix with eleven column headings and a variable number of rows, each with an identifier. Row identifiers are host names that are displayed in a column along the left side of the matrix. Only hosts that have reported exceptions are shown. Initially, you see no host names.

The time stamp carried by the last received exception for each host is displayed beside the host name. Along the top of the matrix the 11 column headings each represent one of the 11 possible “exception severity codes” that can be associated with an exception when it is defined to **filtd**. Even though the **filtd** daemon refers to the codes as describing severity, in reality they are no more than identifiers. Because of this, the **exmon** program refers to the code of an exception as the exception identifier (or exception ID).

As exception packets arrive from remote systems, the matrix starts to become populated. When an exception arrives, the first action is to determine the host name of the host that generated the exception. If the host name is not already displayed in the matrix, the existing rows are moved down and the new host name line is added as the top row. If the host name is already displayed, it is moved to the top of the matrix, pushing all others down.

The matrix cell corresponding to the host that generated an exception (now at the top of the list) and the exception ID of the exception is determined. The cell contains the count of exceptions with this exception ID from the named host. The count is incremented by one to always contain the total number of such exceptions received in the lifetime of this execution of **exmon**.

© Copyright IBM Corp. 1994, 2004

Matrix cells are assigned colors depending on a coloring scheme as specified in the **exmon** X resource file and the exception count in each matrix cell. This is described in "Coloring Scheme" on page 90.

## The exmon Main Window Menu Bar

The menu bar in the main window defines the following pull-down menus:

**File**
This pull-down menu has the following selections available:

**Read Log**
Allows the user to select an exception log to view. An exception log file exists for each host that has sent exception packets to **exmon**. See "Working with Exception Logs" for the placement and naming conventions for exception log files.

**Delete Log**
Allows the user to select an exception log to delete.

**Exit exmon**
The user selects this entry to stop collection of exception packets and to exit the **exmon** program.

**Hosts**
This pull-down menu has the following entries:

**Add Hosts**
When this entry is selected, **exmon** displays a selection list that contains the names of hosts on the network. The list doesn't include hosts that are already being monitored. From the list, the user can select additional hosts to monitor for exception packets.

**Note:** With the Performance Toolbox Local feature, available with Version 2.2 or later, only the local host is available for selection.

**Delete Hosts**
When this entry is selected, **exmon** displays a selection list of hosts that are currently being monitored. The user can terminate the monitoring of exceptions from one or more hosts by selecting hosts from the list.

**Help**
A pull-down menu with the following choices:

**On Version...**
Displays the standard "About" window for **exmon**.

**Help Index**
Displays a list of all help topics in the **exmon** simple help file.

## Working with Exception Logs

As **exmon** receives exception packets from hosts, it adds a line to the host exception log files for those hosts. An exception log is created for each host that sent exception packets to **exmon**.

Exception logs are named after the host that sent the exception packet and have an extension of "**.log**". Exception logs are kept in the directory **$HOME/FiltLogs** for each **exmon** user. For example, the exception log file of user *donnau* that contains exceptions from host **snook** would be **/home/donnau/FiltLogs/snook.log**.

If **exmon** is active more than once for a user at the same time, each active exmon will add exceptions to the log files of hosts as exceptions are received. This causes the log files to contain multiple copies of exception packets. To avoid this, never run multiple copies of **exmon** or make sure each copy is monitoring different hosts.

### Viewing an Exception Log

A user selects the **Read Log** entry from the File pull-down menu of the main window to view an exception log file. A file selection window is displayed with a list of all the exception log files available. The exception log to view is selected from the list in the selection window. After selection of the exception log file, the Show File window is displayed. This window displays the entries in the exception log file in five columns:

1. An action code to indicate any action the user takes for the exception log entry.
2. The name of the exception.
3. The exception's severity code.
4. The date and time the exception was generated.
5. The description of the exception.

The exception log entries are shown in a scrollable list. One or more lines in the list can be selected. When a line has been selected it can either be marked as read, or for deletion. This is done from the menu bar at the top of the Show File window. This menu bar has two pull-down menus:

**File**
: Allows you to select from the following:

**Mark for Delete**
: If the user has selected any line from the scrolled list and then selects this item, an *X* is displayed at the beginning of the line to mark it ready for deletion. When the user saves the changes and closes this window, the selected lines are deleted from the exception log file.

**Mark for Read**
: If the user has selected any line from the scrolled list and then selects this item, an @ symbol is displayed at the beginning of the line to mark it has been read. When the user saves the changes and closes this window, the selected lines are rewritten to the exception log file with an @ symbol at the beginning. The next time the user views the exception log file, the @ symbol is displayed so the user knows the exception has been looked at before.

**Save & Quit**
: If the user selects this item, two things are done. First, if any line has been marked as read, then an @ symbol is placed at the beginning of that line in the exception log file. Second, if any lines have been marked for deletion, those lines are deleted from the exception log file. The Show File window then is closed.

**Quit**
: The Show File window is closed. If any line has been marked as read or for deletion, that information is not saved.

**Help**
: **Index** A help menu containing only the **Help Index** entry.

## Deleting an Exception Log

A user selects the **Delete Log** entry from the File pull-down menu of the main window to delete an exception log file. A file selection window is displayed with a list of all the exception log files available. The exception log to delete is selected from the list in the selection window.

After selection of the exception log, the user must select **OK** to delete the exception log file and refresh the selection window. To close the file selection window, select **Cancel**.

## Working with Hosts

Selection of hosts to monitor for exceptions is done from a selection window. This window has a list of hosts to select from and, at the top, a button with the text **Click here when selection complete**. Similar looking selection lists are used to add hosts to the ones already being monitored and to remove hosts that are no longer to be monitored.

**Note:** With the Performance Toolbox Local feature, available with Version 2.2 or later, only the local host is available for selection.

In either case, select the hosts from the displayed list by moving the mouse pointer to the first host you want, then press the left mouse button and move the mouse while holding the button down. When all hosts you want are selected, release the mouse button. If you want to select hosts that are not adjacent in the list, press and hold the Ctrl key on the keyboard while you select. When all hosts are selected, release the key. After all selections have been made, use the left mouse button to click on the button at the top of the window.

The host selection window is first displayed when you start **exmon**. From this first window you select the initial list of hosts to monitor. When you later want to add or delete hosts, use the Hosts pull-down menu in the main window and select **Add Hosts** or **Delete Hosts** as required.

## Add Hosts

The user is presented with a selection list of all hosts on the network that have responded to invitations from **exmon**. If a host you want to monitor does not show up in the initial selection window, it might be because the host is down, because its **xmservd** daemon cannot be activated, or because its **xmservd** daemon doesn't respond fast enough. In either case, the Add Hosts selection window can be used to refresh the list of hosts and, potentially, make it possible to select the host later.

**Note:** With the Performance Toolbox Local feature, available with Version 2.2 or later, only the local host is available for selection.

## Delete Hosts

If for some reason you don't want to continue the monitoring of one or more hosts, you can stop monitoring them by selecting them from the Delete Hosts window. This window shows a list of the hosts that are currently being monitored by **exmon**. When you select a host from the list and click on the **Click here when selection complete** button, no more exceptions are received from that host. The selected host also is deleted from the matrix in the main window and its exception log files are closed but retained.

## Resynchronizing

Every five minutes, **exmon** attempts to reconnect to any hosts that has dropped out for one reason or another.

## Duplicate Hostnames

When **exmon** generates the list of hostnames available for monitoring, it uses the **RSi** API to solicit for host responses on the network. Every response to this solicitation carries the **uname** (simple, unqualified hostname) of the responding host and the IP address of the host. Because the hosts respond by sending their **uname**, identical hostnames will be returned for each network adapter that received the solicitation for hosts with multiple adapters. This requires special action in **exmon** to prevent exceptions from a host with multiple adapters from being counted multiple times.

In addition to the situation where hosts have multiple adapters, the **uname** usage could also create confusion when two or more hosts have the same **uname**, which could be because the hosts exist in different IP domains or because a host had its simple hostname changed but not its **uname**.

The **exmon** program provides ways to handle these conflicts. The method used depends on the implementation of nameservice in your network. The term *nameservice* means the mechanism that is used to find the IP address or hostname in your network, this can be done with the **host** command. The nameservice is usually provided by a domain name server, by Network Information Services (NIS, formerly Yellow Pages), by the **/etc/hosts** file, or by a combination of these.

The following example shows how different situations are handled. They are based on a network with the following hosts, some of which have multiple network adapters:

| Host uname | IP address | Hostname from nameservice |
|---|---|---|
| banana | 9.11.22.33 | banana.west.com |
| banana | 130.4.5.6 | undefined |
| banana | 192.10.10.10 | banana.west.com |
| banana | 7.7.7.7 | banana.east.com |
| orange | 9.11.22.44 | lemon.west.com |

| Host uname | IP address | Hostname from nameservice |
|---|---|---|
| orange | 9.11.22.55 | orange.west.com |
| lime | 7.8.8.8 | lime.east.com |
| lime | 9.11.22.66 | lime.west.com |

Assuming all these hosts and all their network interfaces receive and respond to solicitation, the list of hosts presented by **exmon** will look like this:

```
130.4.5.6 (banana)              130.4.5.6
banana.east.com                 7.7.7.7
banana.west.com                 9.11.22.33
lemon.west.com                  9.11.22.44
lime.east.com                   7.8.8.8
lime.west.com                   9.11.22.66
orange.west.com                 9.11.22.55
```

For the host with **uname** banana, the adapter with IP address `192.10.10.10` does not show up. That is because the hostname is exactly the same as for the adapter with IP address `9.11.22.33`. This generally means that there are two network adapters on the same machine. The IP address chosen is what the nameservice returns when you do a name lookup on the hostname. The adapter whose IP address was not defined with the nameservice shows up with its IP address followed by the **uname** of the host. Finally, the adapter that has a different host/domain name assigned to it shows up on a line by itself.

Because all of banana's adapters received the solicitation and responded to it, exception packets will be sent once on each interface. Consequently, **exmon** will receive four exception packets for each exception that occurs on banana. The one received from IP address `192.10.10.10` is discarded because **exmon** was able to determine that the host was the same as the host with IP address `9.11.22.33`. All other exception packets will be processed by **exmon** provided they have elected to monitor the host entry.

In all the other cases, **exmon** will detect that the seemingly identical hosts (judging from their unames) are indeed entities that should be treated as separate. In general, unless there are compelling reasons to assign different hostnames to multiple interfaces in one host, it is an advantage to explicitly assign the same hostname to all interfaces. Whether you select different hostnames for each adapter or not, register each IP address with the nameserver.

## Command Execution from exmon

As an exception monitoring program, **exmon** has the function to alert its user if some exception is reported from one of the monitored hosts. When this happens, the user might want to look into the causes of the exception.

To facilitate this, the **exmon** program gives the user the ability to execute commands for any of the hosts that reported exceptions. Commands are defined in a configuration file, **exmon.cf** as explained below. To execute a command for a host, move the pointer to one of the host names displayed in the **exmon** monitoring window and click the left button. This causes a dialog box to pop open and display the commands the user can execute for that host. To execute a command, the user selects one of the lines and then selects **OK**.

The list of commands that are displayed in the dialog box is kept in the user definable **exmon.cf** file. The **exmon** program first looks for this file in the user's home directory. If the file is not found there, it is looked for in the **/etc/perf/** directory. If the file is not there, then the **/usr/lpp/perfmgr** directory is searched. If the file cannot be located in any of the directories, the program will be missing important information and may terminate or provide reduced function. A sample configuration file is provided in the **/usr/lpp/perfmgr** directory.

## The exmon Configuration File

The configuration file may contain comment lines that begin with the character # (number sign). All other lines are assumed to define commands and have the general format:

```
command_name:command_line
```

| | |
|---|---|
| `command_name:` | The command_name is what appears in the dialog box when a user selects the host name. It should explain the function of the command in enough detail for a user to determine which of the defined commands is appropriate for the situation. The command_name must be followed by a colon. |
| `command_line` | At the places where a host name is required in the command line, the characters %s (percent sign followed by a lowercase S) must be present in the command line. The %s may be included in the command line a maximum of five times. |

The configuration file may contain a maximum of 50 commands. Each command line should end with an ampersand (&) character. This executes the command in the background. If the command is not executed in the background, then **exmon** execution is suspended until the command has terminated.

The following is an example of an exmon configuration file:

```
#Sample Exmon Config File
#Display processes for host with 3dmon
3dmon Processes:3dmon -h%s -cProc &
#chmon for selected host
chmon:aixterm -n chmon -T chmon -e ksh -c "(sleep 1; chmon %s)" &
#Start xmperf for host
xmperf:xmperf -h%s &
#xmpeek statistics
xmpeek:aixterm -n xmpeek -T xmpeek -e ksh -c "(xmpeek %s; read)" &
```

## The exmon Resource File

The AIXwindows resource file for **exmon** defines resources you can use to enhance the appearance and behavior of **exmon** and is installed as **/usr/lib/X11/app-defaults/EXmon**. The following behavior can be modified with the **exmon** resource file.

## Coloring Scheme

The user decides whether a separate color should be assigned to the cells belonging to each of the exception IDs, or whether the cells of all exception IDs should follow a common coloring scheme that assigns colors depending on the number of exceptions received in each of the cells. To select the first coloring scheme, set the X resource **RangeDisplay** to false, otherwise set it to true. If the resource is set to true, the coloring scheme in the section entitled “Value Ranges” is in effect; otherwise the coloring scheme described in “Exception Colors” is used.

### Exception Colors

Each exception identifier has a different color associated with it. Colors are defined through the X resources **ValueColor0** through **ValueColor10**.

### Value Ranges

The user can define the exception monitor to change colors when the number of exceptions for an individual exception ID is within a certain range. The definition of color ranges applies across all exception IDs. Two limits between value ranges are defined with the X resources **ValueRange1** and **ValueRange2** to create a total of three ranges. The X resources **RangeColor1** through **RangeColor3** are used to define the colors to use when the exception count in a grid cell falls within a range. To illustrate, consider this example where a user defines the following resources:

```
*RangeDisplay: true
*ValueRange1: 5
*ValueRange2: 10
*RangeColor1: green
*RangeColor2: yellow
*RangeColor3: red
```

With this resource setting, if the number of exceptions for a particular exception ID is less than or equal to 5, then the color displayed is green. When the number of exceptions is within the range 6-10, then the color displayed is yellow. When the number of exceptions has increased beyond 10, then the color displayed is red.

## Exception Identifier Text

The **filtd** daemon allows you to define severity codes for each alarm. These codes are sent as part of any generated exception packets. Obviously, these codes and exceptions can be defined to represent anything you want them to represent. By setting the **ExceptionText0** through **ExceptionText10** resources, you can define the column headings displayed in the **exmon** main window for the exception IDs. A maximum of seven characters is displayed for each column heading.

# Chapter 9. Recording Files, Annotation Files, and Recording Support Programs

This chapter explains the contents of recording files and how the files can be created and processed. It then describes each of the following programs collectively known as the recording support programs:

**a2ptx**
A program to generate recordings from ASCII files.

**ptxmerge**
A program to merge up to 10 recording files into one.

**ptxsplit**
A program to split recording files into multiple files

**ptxconv**
A program to convert between Performance Toolbox for AIX Version 1.1 to Version 2 or Version 1.2 recording file format.

**ptxtab**
A program to tabulate the contents of recording files.

**ptxls**
A program to list the statistics in a recording file.

**ptxrlog**
A program to create ASCII or binary recording files.

**ptx2stat**
A program to convert binary recording files containing hotset information.

**ptxhottab**
A program to tabulate hotset information in recording files.

The main program for analyzing recordings is the **azizo** program described in Chapter 10, “Analyzing Performance Recordings with azizo,” on page 107.

**Note:** You can access these programs from the xmperf Command Menu Interface.

## Recording Files

Recording files are binary files whose first record is a configuration record. This record identifies the file as a recording file, names the source of the recording, and states the version of the file. To be valid, recording files must also contain the definition of one or more statistics and must contain at least one value record.

## Creation of Recording Files

Recording files are created by one of the Agent or Manager programs. They can be created by the **xmperf** and **3dmon** programs during monitoring, by the **xmservd** daemon at any time it is running, by the program **a2ptx** from ASCII files that adhere to a certain format, or by the **ptxrlog** program.

From the definition of statistics, a program reading the recording file can determine how the statistics are grouped into sets of statistics (statsets). By the nature of the record layout, at least one such set exists but the definition records may define multiple. Sets of statistics are defined by the program that creates the recording:

**xmperf**
A set is created for each instrument in the console from which the recording is created.

**3dmon**
A set is created for each path name at the right side of the graph grid.

**xmservd**
A set is created for each sampling interval. Each statset is assigned a number equal to the sampling interval divided by the minimum sampling interval of the **xmservd** daemon.

**a2ptx**
Only one set is created.

**ptxrlog**
Only one set is created.

Recordings created by **xmperf** also contain console definition records. This record type describes the layout and other properties of the console that was used to create the recording and is used by **xmperf** to reconstruct the console when the recording is played back by **xmperf**. Recordings created by other

© Copyright IBM Corp. 1994, 2004

programs do not contain console definition records. When such recordings are played back with **xmperf**, default consoles are constructed as explained in the section entitled "Creation of Playback Consoles" on page 55.

Records carrying the observations are called *value records*. They correspond to the sets defined in the recording file and contain one reading for each of the statistics in the set plus the time stamp of the reading and the elapsed time since the previous reading. Each reading consists of two fields:

**Raw value**
Regardless of the type of statistic, gives the actual value as observed. Usually, programs that process a recording file use this field for statistics of type *SiQuantity*. Such statistics represent a level, such as the amount of free space on a disk or the percentage of system memory in use.

**Delta**
The difference between the previous observation for this statistic and the latest observation. Usually, programs that process a recording file use this field for statistics of type **SiCounter** and divide it with the elapsed time in seconds to arrive at a rate per second. **SiCounter** statistics represent a count of activity such as the number of disk operations or the number of timer ticks while the CPU is idle.

A special type of value record is the *stop record* which signals that recording was stopped for a statset and gives the time it happened. This allows programs using the recording file to distinguish between gaps in the recording and variances in recording interval.

## Modifying Recording Files

Several programs can modify recording files. As they do so, they may preserve or discard information about sets of statistics and consoles in the files. The following sections describe how recording files can be modified. More detail is provided in the detailed description of each of the programs later in this chapter and in Chapter 13, "Monitoring Remote Systems," on page 153.

### Filtering with azizo

The **azizo** program allows you to write a filtered recording file from an input recording file. Filtering can write a subset of statistics for a subset of the time span covered by the input recording file. If the input recording file contains more than one definition of sets, or if it contains a console definition, you can elect to discard these definitions and create the filtered file with only one set and no console definition. Creation of filtered recording files with the **azizo** program is described in the article entitled "Filtered Recordings" on page 122.

### Merging with ptxmerge

The program **ptxmerge** allows you to merge multiple recording files into one. When merging only two recordings, the program can be asked to adjust the time stamps of one of the recordings. If the multiple recordings you merge into one all contain identical console definitions, the console and set definitions are retained. Otherwise, all console definitions are discarded while set definitions are retained.

The **ptxmerge** program also allows you to reorganize recording files where multiple recordings are concatenated into one file. It does so by first splitting the files into separate files, then merging them together.

### Splitting with ptxsplit

Very large recording files can be time consuming to analyze. The program **ptxsplit** can be used to divide such files into smaller files. Splits can be done as simply as dividing the file into sections where the only change to the sequence and contents of records is that each output file has a copy of the configuration and definition records.

More advanced features allow you to split the file into groups of selected statistics. When this is done, you have the option of preserving or discarding definition records, depending on how you have specified the split to take place. For some splits, it may not be possible to preserve the definition records.

### Version Conversion with ptxconv

Recording files created by Version 1.1 of the Performance Toolbox for AIX can be converted to the format used by Versions 1.2 or 2. If so desired, the program also does conversion the other way. Recording files created by Version 1.1 do not have the special stop records and neither do files converted from Version 1.1 format. When you analyze such files with **azizo**, a stop in the recording is interpreted as an extraordinarily long sampling interval.

## Annotation Files

**Note:** Annotation files are available with Version 2.2 or later only.

Annotation files are plain ASCII text files. They are associated with recording files only through a naming convention. For example, the annotation file for a recording file named **R.time_off** would be **N.time_off.** If the name of a recording file is changed to something that does not begin with the **R.** prefix, the association with the annotation file is lost, even if the annotation file is renamed.

If the recording file is moved to a different directory and the annotation file is not moved to the same directory, then the association with the annotation file is lost.

If the recording file is processed with the recording support programs, then the modified (and differently named) new recording file will no longer have any association to the annotation file. However, none of the recording support programs change the original recording file so the association to those files still exists.

Annotation files can be created and modified from **xmperf**, **3dmon**, **3dplay**, and **azizo**. From **xmperf** and **3dmon**, annotation files can only be created and modified if recording is, or has been, activated. From **3dplay** and **azizo**, the user can create or modify annotation files at any time.

## The a2ptx Recording Generator

The **a2ptx** program takes a file with a tabulated list of data as input and produces a recording file in a format that allows the file to be processed by any recording support program and by **xmperf** and **azizo**. The purpose is to extend the usability of Performance Toolbox for AIX to cover other types of data or performance data produced by programs that are not part of Performance Toolbox for AIX.

## Input Formats of a2ptx

For **a2ptx** to successfully create a valid recording file from an input file, the latter must be in a certain format. When used with their **-s** command line flags, the programs **ptxtab** and **ptxrlog** produce output in a format suitable for **a2ptx**. For **ptxrlog**, the **-t** flag can also be used. There are rules for the following parts of the input file:

- "Host identifier"
- "Statistic names" on page 96
- "Time stamps" on page 96
- "Data values" on page 96.

For the following explanation, refer to an "ptxspread" on page 102, which shows an example of a valid **a2ptx** input file.

### Host identifier

If a string in the format *hostname: xxxxx* is at the end of the first line of the input file, this causes **a2ptx** to prefix all statistic names it later reads with the string *hosts/xxxxx*. The keyword *hostname:* may start with a uppercase *H* and must be followed by a colon. The *xxxxx* part can be any value you want. The two parts must be separated by white space. If the host name string is not found on the first line of the file, no prefix is added to the statistic names.

### Statistic names

Before any time stamps or data values appear in the input file, a line beginning with the string"Timestamp" must exist. The double quotation marks are optional. Following the identifier string on the line must be one column heading for each column of data values on data value lines. The column headings may optionally be enclosed in double quotation marks and are used as the fully qualified name of the statistic records written to the output recording file. The line's identifier string must be separated from the column headings with white space, which must also separate the column headings.

### Time stamps

Each line with data values must begin with a time stamp optionally enclosed in double quotation marks and followed by white space. The format of the string is:

*YYYY/MM/DD hh:mm:ss*

Time stamps should appear in ascending order to make the resulting output file usable in other Performance Toolbox for AIX programs.

### Data values

Following the time stamp on data lines must be a number of data values separated by white space. Data values may have a decimal point or be integers. The number of data values on each line must be the same as the number of column headings. If one or more data lines do not have data available for a value, a dash must be inserted in place of the missing data value.

## The a2ptx Command Line

The command line to invoke **a2ptx** is:

**a2ptx** *input_file output_file*

Both arguments are required.

## The ptxmerge Merge Program

The **ptxmerge** program has two modes of operation:

**Rearrange** To rearrange the sequence of records in a recording file that contains more than one set of control information.

**Merging** To merge multiple recording files into one.

When you supply only one input file name on the **ptxmerge** command line, it is assumed that you want to rearrange the records in that file. In all other cases, merging is assumed.

The actual implementation of the rearrange function divides the input file into separate temporary files, one for each set of control information in the file. Those temporary files are then merged into one to create the final output file and deleted.

The created output file always has only one group of control records at the beginning of the file. This set is created from the groups of control records in the input files. If all input files were created by **xmperf** and have identical definitions (were created from identically configured consoles that contain identical instruments), all control records, including the console definition, are preserved and written to the output file.

If there's the slightest difference between the control records of the input files, then **ptxmerge** creates the output file so that any console information is discarded. Definitions of sets of statistics (statsets) is then only retained if the **-z** command line argument is used. In all other cases, the resulting output file contains only a single set of statistics (statset).

The value records that carry observations of statistics are arranged so they appear in time order in the resulting output recording file.

## When to Use ptxmerge

The **ptxmerge** program is safe to use because it always leaves the original files unchanged and it is reasonably fast in doing what it does. It can be used to organize recordings for optimal analysis or playback, and it works in concert with the other recording support programs as well as the filtering function of **azizo**. The following two sections give some examples of when to use **ptxmerge**.

### Rearranging Recording Files

The **xmservd** daemon can produce recording files with more than one set of control information. It happens whenever **xmservd** resumes recording to an existing recording file and the **xmservd** recording configuration file (**xmservd.cf**) has been changed since the recording file was created. It is done to prevent the recording from being corrupted when changes are made to the sets of statistics being recorded.

The **xmservd** daemon makes no attempt at keeping track of what the changes to its configuration file were, so it is possible that the two or more sets of control information in the recording file are identical. On the other hand, there's no guarantee that they are. The **ptxmerge** program allows you to rearrange the file and detects if the sets are indeed identical. If they are, the resulting output file are identical to the input file, except that only the first set of control records is preserved.

If a recording file produced by **xmservd** has different sets of statistics, or if a recording file was produced by concatenating two or more recording files with different sets of statistics, then the sets are merged into one single set of statistics unless the **-z** command line argument is used.

The only other program that can create recording files with more than one set of control information is **ptxrlog**. See "The ptxrlog Recording Program" on page 103 for more information.

### Merging Recording Files

There can be many reasons to merge one or more recording files into one. Generally, it is done because you want to analyze multiple recordings as one with **azizo** or because you want to play multiple recordings back as one with **xmperf**. The separate recording files may represent recordings from the same invocation of **xmperf**, but from different instruments. They may be produced simultaneously on different hosts, possibly to record the effects of a distributed application on the application's server and client sides. Or they may represent identical recordings for different time periods that you want to analyze together.

If the intention is to play the recordings back with **xmperf**, then it is often a good idea to use the **-z** flag to preserve instrument definitions. This allows you to keep track of the original sets of statistics, which is especially important if you use **ptxmerge** to adjust the time stamps of one of the input files.

## The ptxmerge Command Line

The **ptxmerge** program allows the user to specify up to 10 input files that are to be merged into one file. All files must be valid Performance Toolbox for AIX recording files in Version 2 format. When more than one input file is specified and one or more of the input files contain multiple sets of control information, only the records belonging to the first such set participate in the merge operation.

If only one input file is given, the program assumes you want it to rearrange the records in that file. If this file contains only one set of control information, then the output file is identical to the input file.

The command line to invoke **ptxmerge** is:

**ptxmerge** [ **-m**| **-p** *incr*|-t inc *r*] [ **-z**] *outfile input1* [ *input2* [ *input3*...]]

The command line flags have the following meaning:

**-m** Only valid if exactly two input files are specified. Merges files, modifying all time stamps in the oldest file by the difference in time between the time stamps of the first value record in the two files.

**-p** Only valid if exactly two input files are specified. Must be followed by the number of seconds to be added to all time stamps in the first input file before merging the files. This value may be negative.

**-t** Only valid if exactly two input files are specified. Must be followed by the number of seconds to be added to all time stamps in the second input file before merging the files. This value may be negative.

**-z** *Optional.* Preserves information about sets of statistics (statsets) when creating the resulting file. This is useful if the output file is to be used for playback with **xmperf**. The input files are merged together but each set of statistics are played back in instruments of the same contents (though not necessarily the same appearance) as the originals.

## The ptxsplit Split Program

At times, it is advantageous to divide one recording file into multiple. This may be because the file is too large to allow timely analysis or playback, because it contains statistics that are irrelevant for the current use of the file, or because it contains more than one set of control records. In either case, **ptxsplit** splits a recording file to your specifications.

## The ptxsplit Command Line

The **ptxsplit** program is invoked with the following command line:

**ptxsplit** { **-p** *parts*| **-s** *size*| **-h**| **-b**| **-f***cfile*| **-d** *hhmm* [ **-t** *dhhmm*]} *infile*

The command line arguments are all mutually exclusive, except that the **-t** argument is only valid if the **-d** argument is given. One of the arguments must be specified. The arguments are:

**-p** Split in parts of equal size. Must be followed by the number of parts the input file shall be divided into. The output files are approximately the same size and begin with a set of control records. The output file names are **infile.p1, infile.p2, ... infile.pn**. Statsets are preserved in the output as are any console records.

**-s** Split in parts of equal size. Must be followed by the size you want each output file to have. The output files, except the last one, usually are slightly smaller than the specified size; the last file may be much smaller. The output files all begin with a set of control records. The output file names are **infile.s1, infile.s2, ... infile.sn**. Statsets are preserved in the output as are any console records.

**-h** Split into files according to the host name of individual observations. The output files all begin with a set of control records. The output file names are **infile.hostname1, infile.hostname2, ... infile.hostname**. Statsets are preserved in the output. Any console records are discarded.

**-b** Split into files for each set of control records encountered. The output files all begin with a set of control records. The output file names are **infile.b1, infile.b2, ... infile.bn**. Statsets are preserved in the output as are any console records.

**-f** Split into two files. The flag must be followed by a file name of a control file. The first output file is to contain all occurrences of the statistics listed in the control file. Remaining statistics are written to the second output file. Statistics are specified in the control file with their full path name. The control file may contain comment lines beginning with the character # (number sign). If the **hosts**part of the path name is omitted, statistics are selected across all host names. If the **hosts**part of the path name is supplied, an exact match is required for a statistic to be selected. The first output file has the name **infile.sel**, the second outfile is called **infile.rem**. Statsets are not preserved in the output files.

The program **ptxls** can produce a list of the statistics contained in a recording file. The output from the program has the format required for the control file. Use it by redirecting **ptxls** output to a file; then edit the file to include only the statistics you want in the file **infile.sel**.

**-d** Split after duration into parts covering time periods of equal size. Must be followed by the duration span of each file, given as **hhmm**, where:

*hh* = Hours.

*mm* = Minutes.

If the **-t** argument is omitted, the time period begins with the earliest value record in the input file; otherwise with the time specified on the **-t** argument. The output files all begin with a set of control records. The output file names are **infile.d1, infile.d2, ... infile.dn**. Statsets are preserved in the output as are any console records.

**-t** Only valid if the **-d** argument is given. Specifies a point in time that shall be used to split the input file. Must be followed by a time in the format **dhhmm**, where:

*d* = Day of week, Sunday = day 0.

*hh* = Hours.

*mm* = Minutes.

The time given may lie outside the time period covered by the input recording file. If the time given differs from the time stamp of the first value record in the input file, the first output file contains data for an interval smaller than that requested with the **-d** argument.

For example, assume a recording file's first value record has a time stamp corresponding to 30830 (day 3, at 8:30 a.m.) and you invoke **ptxsplit** with the command line:

```
ptxsplit -d0600 -t00000 recording_file
```

This causes the first file to cover the interval from 8:30 a.m. until 11:59 a.m., the next one from 12:00 noon until 5:59 p.m., and so on until there's no more value records in the input file.

Consider splitting the same file with the command line:

```
ptxsplit -d0600 -t40800 recording_file
```

The **-t** argument, in this case, gives a point in time later than the first value record's time stamp. The program determines the time to place the first split point by stepping backwards in time from day 4 at 8:00 a.m. in steps of six hours (as per the **-d** argument) until it has passed the time stamp of the first value record. This would be on day 3 at 8:00 a.m. This is the reference point. The first output file covers day 3 from 8:30 a.m. to 1:59 p.m., the next from 2 p.m. to 7:59 p.m., and so forth.

## The ptxconv Conversion Program

In Version 1.1 of the Performance Toolbox for AIX, recording files could only be created with the **xmperf** program. The only other program in Version 1.1 that used recording files was the **xmtab** program, which is renamed **ptxtab** in later versions of Performance Toolbox for AIX. Neither program was concerned with the possibility of a recording file being created from a monitoring session where the recording of instruments was stopped and started one or more times. Similarly, neither program ever attempted to read a recording file backwards.

This changed with the introduction of the **azizo** program and the recording support programs. The **azizo** program wants to know if a recording session was stopped and restarted, and to speed the analysis, the program reads some records off the end of the recording file. Consequently, the recording file format was changed so that stop records are added when a recording stops, and so a recording file can be read backwards. The changes mean that Version 1.1 files cannot be processed by **azizo** or other Version 2 or Version 1.2 Performance Toolbox for AIX programs.

To allow a user of Version 1.1 of Performance Toolbox for AIX to upgrade to later versions without losing the ability to view existing recording files, the **ptxconv** program was added to Performance Toolbox for AIX. This program allows conversion between Version 1.1 and Version 2 and Version 1.2 recording files in either direction.

**Note:** When a recording file has the Version 1.1 format, whether it was created that way or was converted from a later format, a conversion to the later format will not add stop records to the file. This causes **azizo** to treat gaps in the recording of a set of statistics as an extraordinarily long sampling interval.

## The ptxconv Command Line

The **ptxconv** program is invoked with the following command line:

**ptxconv -v**{**1**|**2**} *input_file output_file*

All command line arguments are required and have the following meaning:

**-v1**
Converts a recording file with Version 2 or Version 1.2 format into the Version 1.1 format. This allows the use of an older version of **xmperf** to play the recording back.

**-v2**
Converts a recording file with Version 1.1 format into the later format. This allows any of the Performance Toolbox for AIX Version 1.2 and Version 2 programs that process recording files to work with the converted file.

**input_file**
The path name of a recording file. The input file should be a file that was created with the version level the user wishes to convert from.

**output_file**
The path name the user wishes the new recording file to have.

## Listing Recorded Data with ptxtab

A simple utility program, **ptxtab** lets you format a recording for tabulated output. In earlier versions of Performance Toolbox for AIX, this program was called **xmtab**. The program takes a recording file as input and produces one output file for each set of statistics (statset) in the recording file.

Each of the output files are named by suffixing the statset sequence number to the name of the recording file. If the recording file has an original file name as created by **xmperf** or **3dmon**, the initial ″**R**″. is changed to ″**A**″. to distinguish between Recordings and ASCII output files. This also ensures that ASCII output files do not show up in the dialog window used to select recordings in the programs **xmperf** and **azizo**. As an example, assume the recording file **$HOME/XmRec/R.NiceMonitor** was created by **xmperf** and has instruments (statsets) with sequence numbers 3 and 8. Running the **ptxtab** program would then produce the ASCII output files **$HOME/XmRec/A.NiceMonitor_3** and **$HOME/XmRec/A.NiceMonitor_8**.

By default, each of the output files produced by **ptxtab** is formatted for printing with a multiline heading that begins with a page eject. The first line lists the name of the console (if console information is available in the recording file) or program (when the recording file was not created by **xmperf**) that created the recording and the host name of the host providing the data. The second line is blank and the remaining lines provide headings for each column of tabulated data. The following is an example of output from **ptxtab** as produced with no command line flags.

## Example of ptxtab Default Output Format

```
#Monitor: Nice Monitor --- hostname: nchris

                                                         Mem       Mem
                                PagSp       PagSp        Virt      Virt
Timestamp                       %totalused  %totalfree   pagein    pageout
1994/01/07 15:36:03             27.8        72.2         8         20
```

```
1994/01/07 15:36:07           27.8        72.2        7          17
1994/01/07 15:36:11           27.8        72.2        3          283
1994/01/07 15:36:15           27.8        72.2        28         48
1994/01/07 15:36:19           28.2        71.8        56         41
1994/01/07 15:36:23           29.5        70.5        29         38
1994/01/07 15:36:27           31.5        68.5        0          62
1994/01/07 15:36:31           32.4        67.6        70         1
1994/01/07 15:36:35           32.6        67.4        73         32
1994/01/07 15:36:39           32.8        67.2        156        0
1994/01/07 15:36:43           34.5        65.5        167        4
1994/01/07 15:36:47           34.4        65.6        163        0
1994/01/07 15:36:51           31.1        68.9        12         57
1994/01/07 15:36:55           30.2        69.8        35         34
1994/01/07 15:36:59           28.0        72.0        15         0
1994/01/07 15:37:04           28.0        72.0        15         0
```

## The ptxtab Command Line

The **ptxtab** command line looks as follows:

```
ptxtab [-l lines| -c | -s [-r | -t] recording_file
```

**-l** The flag **-l** (lowercase L) is used to specify the number of lines per page you want the output files formatted for. The default is 23 lines per page, which is ideal for viewing the output in a 25-line window or on a terminal with 25 lines. If you specify 0 (zero) lines per page, pagination is suppressed. If the value is given as non-zero, it must be between 10 and 10,000. The flags **-l**, **-c**, and **-s** are mutually exclusive.

**-c** The flag **-c** causes **ptxtab** to format the output files as comma separated ASCII. Each line in the output files contains one time stamp and one observation. Both fields are preceded by a label that describes the fields. An example of output formatted this way is shown in the“Example of ptxtab Comma-Separated Output Format.” The eight detail lines shown correspond to the first two detail lines in the “Example of ptxtab Default Output Format” on page 100. The flags **-l**, **-c**, and **-s** are mutually exclusive.

**-s** The flag **-s** causes **ptxtab** to format the output files in a format suitable for input to spreadsheet programs. When this flag is specified, it is always assumed that the **-r** flag is also given. An example of formatting with the **-s** flag is shown in the “ptxspread” on page 102. The detail lines shown correspond to the detail lines in the “Example of ptxtab Default Output Format” on page 100. This output format also matches the requirements of the **a2ptx** input file format. The flags **-l**, **-c**, and **-s** are mutually exclusive.

**-r** The flag **-r** is independent of the other flags. It specifies that when *SiCounter* data is sent to the **ptxtab** output files, they are presented as rates per second. Without this option, **ptxtab** presents this data as the delta value in the interval. The flags **-r** and **-t** are mutually exclusive.

**-t** The flag **-t** is independent of the other flags. It specifies that when SiCounter data is sent to the **ptxtab** output files, they are presented as absolute values. In other words, this flag causes SiCounter values to be treated as SiQuantity values. Without this option, **ptxtab** presents this data as the delta value in the interval. The flags **-r** and **-t** are mutually exclusive.

## Example of ptxtab Comma-Separated Output Format

```
#Monitor: Nice Monitor --- hostname: nchris
Time="1994/01/07        15:36:03",       PagSp/%totalused=27.82
Time="1994/01/07        15:36:03",       PagSp/%totalfree=72.18
Time="1994/01/07        15:36:03",       Mem/Virt/pagein=8
Time="1994/01/07        15:36:03",       Mem/Virt/pageout=20
Time="1994/01/07        15:36:07",       PagSp/%totalused=27.82
Time="1994/01/07        15:36:07",       PagSp/%totalfree=72.18
Time="1994/01/07        15:36:07",       Mem/Virt/pagein=7
Time="1994/01/07        15:36:07",       Mem/Virt/pageout=17
```

## ptxspread

```
#Monitor: Nice Monitor --- hostname: nchris
"Timestamp"              "PagSp/       "PgSp/        "Mem/Vir/     "Mem/Vir/
                          %totused"     %totfree"     pagein"       pageout"
"1994/01/07  15:36:03"     27.8          72.2          8             20
"1994/01/07  15:36:07"     27.8          72.2          7             17
"1994/01/07  15:36:11"     27.8          72.2          3             283
"1994/01/07  15:36:15"     27.8          72.2          28            48
"1994/01/07  15:36:19"     28.2          71.8          56            41
"1994/01/07  15:36:23"     29.5          70.5          29            38
"1994/01/07  15:36:27"     31.5          68.5          0             62
"1994/01/07  15:36:31"     32.4          67.6          70            1
"1994/01/07  15:36:35"     32.6          67.4          73            32
"1994/01/07  15:36:39"     32.8          67.2          156           0
"1994/01/07  15:36:43"     34.5          65.5          167           4
"1994/01/07  15:36:47"     34.4          65.6          163           0
"1994/01/07  15:36:51"     31.1          68.9          12            57
"1994/01/07  15:36:55"     30.2          69.8          35            34
"1994/01/07  15:36:59"     28.0          72.0          15            0
"1994/01/07  15:37:04"     28.0          72.0          15            0
```

# The ptxls List Program

The **ptxls** program allows you to list the control records of a recording file. You can use it to find out if more than one set of control records exist, and it tells you which statistics are contained in a recording file. In addition, it lists the time period covered by the recording and the number of value records (observation count) for each set of control records in the file.

The output created by **ptxls** is suitable for use with the **-f** option of the **ptxsplit** program. That option requires a file name of a file that contains a list of statistics to extract from a recording. This file can be created by redirecting the output of **ptxls** to a file and then deleting the path names that should not be extracted.

The **ptxls** command line takes no flags but you must specify the name of a recording file to process.

The first output line names the set of control records in the line displaying `Configuration`. Then, each instrument in the console is identified by one line displaying `Console` giving the number assigned to the instrument plus one line for each of the statistics belonging to that instrument (statset). Each statistic line ends with two numbers. The first identifies the statset and the second is the statistic identifier within the statset.

The following is an example of **ptxls** output. The example is created from a recording of the **xmperf** console ″Combo Style Sample″ as supplied in the sample **xmperf** configuration file:

```
# Configuration: ID=Combo Style Sample
#   Console   #00001:    ID=Combo   Style   Sample   hosts/nchris/SysIO/writech
00001/00002          hosts/nchris/SysIO/readch
00001/00003
#   Console   #00002:    ID=Combo   Style   Sample   hosts/nchris/Mem/Real/%local
00002/00001          hosts/nchris/Mem/Real/%clnt
00002/00003          hosts/nchris/Mem/Real/%free
00002/00005
#   Console   #00006:    ID=Combo   Style   Sample   hosts/nchris/Disk/hdisk0/xfer
00006/00001          hosts/nchris/Disk/hdisk1/xfer
00006/00002          hosts/nchris/Disk/hdisk2/xfer
00006/00003          hosts/nchris/Proc/pswitch
00006/00005          hosts/nchris/Proc/runque
00006/00007
#   Console   #00003:    ID=Combo   Style   Sample   hosts/nchris/CPU/cpu0/wait
00003/00001          hosts/nchris/CPU/cpu0/kern
00003/00002          hosts/nchris/CPU/cpu0/user
00003/00004          hosts/nchris/Syscall/total
```

```
00003/00005
#   Console   #00008:    ID=Combo   Style   Sample   hosts/nchris/CPU/cpu0/kern
00008/00001          hosts/nchris/CPU/cpu0/wait
00008/00002          hosts/nchris/Mem/Real/%free
00008/00003          hosts/nchris/PagSp/%totalfree
00008/00004          hosts/nchris/Proc/swpque
00008/00005
#   Console   #00004:    ID=Combo   Style   Sample   hosts/nchris/Disk/hdisk0/busy
00004/00001          hosts/nchris/Disk/hdisk1/busy
00004/00002          hosts/nchris/Disk/hdisk2/busy
00004/00003
#   Console   #00009:    ID=Combo   Style   Sample   hosts/nchris/Mem/Virt/pagein
00009/00001          hosts/nchris/Mem/Virt/pageout
00009/00002          hosts/nchris/Mem/Virt/pgrclm
00009/00003          hosts/nchris/PagSp/hd6/%free
00009/00004          hosts/nchris/Mem/Real/%free
00009/00006
#   Statistics for above:     Start time Wed Jan 12 18:11:19 1994

#                             End time Wed Jan 12 18:20:09 1994

#                             Observation count          1881

#
```

## The ptxrlog Recording Program

The ptxrlog program can produce recordings in either ASCII format, which allows you to print the output or postprocess it with database or spreadsheet programs or with the **a2ptx** program to produce a standard Performance Toolbox for AIX recording file, or it can produce a standard Performance Toolbox for AIX recording file in binary format. Statistics to record are specified from a control file, the command line, or both. If **ptxrlog** is executed in the background, the list of statistics to record must be specified in a control file.

The **ptxrlog** program uses the RSI to access statistics and can collect and record statistics from one host at a time across the network. All statistics are defined in one statset.

## The ptxrlog Command Line

The **ptxrlog** command line looks as follows:

**ptxrlog** {**-f** *infile* | **-m** | **-mf** *infile* } [**-h** *hostname* ] [**-i** *seconds* ] [**-o** *outfile* [**-c** | **-s** | **-t** ] | **-r** *binoutfile* ] [**-l** *pagelen* ] [**-b** *hhmm* ] [**-e** *hh.mm* ]

**-f** Name of a control file that contains a list of statistics to record. In the control file, each statistic must be given on a line on its own and with its full path name, excluding the host part, which is supplied by **ptxrlog** either from the **-h** argument or by using the local host name. If the **-f** argument is not given, the user is prompted for a list of statistics. If both the **-f** and the **-m** arguments are given, **ptxrlog** first selects the statistics given in the control file, then prompts the user to specify additional statistics.

**-m** Manual input of statistic names. The user is prompted for a list of statistic names to be entered as full path names without the host part. The host part is supplied by **ptxrlog** either from the **-h** argument or by using the local host name. If both the **-f** and the **-m** arguments are given, **ptxrlog** first selects the statistics given in the control file, then prompts the user to specify additional statistics.

**-h** Hostname of the host to monitor. This argument is used to identify the host to be monitored and, thus, to create the *hosts* part of the path names for the statistics to monitor. If this argument is not supplied, the host name of the local host is used.

**-i** Sampling interval. Specifies the number of seconds between sampling of the specified statistics. If this argument is not supplied, the sampling interval defaults to 2 seconds.

**-o** Output file name. Specify the name of the output file you want. If this argument is omitted, output goes to standard output and neither of the format flags **-c**, **-s**, or **-t** is permitted. If **-o** is given but neither of the three format flags is, the output looks the same as the output from **ptxtab** shown in the “Example of ptxtab Default Output Format” on page 100. The **-o** flag and the **-r** flag are mutually exclusive.

**-c** The flag **-c** causes **ptxrlog** to format the output file as comma separated ASCII. The flag is only valid if **-o** is given. Each line in the output file contains one time stamp and one observation. Both fields are preceded by a label that describes the fields. The output looks the same as the **ptxtab** output shown in the “Example of ptxtab Comma-Separated Output Format” on page 101. The flags **-c**, **-s**, and **-t** are mutually exclusive.

**-s** The flag **-s** causes **ptxrlog** to format the output file in a format suitable for input to spreadsheet programs. The flag is only valid if **-o** is given. The output looks the same as the output from **ptxtab** output shown in the “ptxspread” on page 102. The flags **-c**, **-s**, and **-t** are mutually exclusive.

**-t** Tab separated format. This flag is identical to the **-s** flag except that individual fields on the lines of the output file are separated by tabs rather than blanks. The flag is only valid if **-o** is given. The flags **-c**, **-s**, and **-t** are mutually exclusive.

**-r** The **-r** flag specifies that the output from **ptxrlog** goes to a binary recording file in standard recording file format. The name of the output file must be specified after the flag. The **-o** flag and the **-r** flag are mutually exclusive.

**-l** (Lowercase L) Specifies the number of lines per page when neither the **-o** nor the **-r** flag is specified or when the **-o** flag is specified but neither of the **-c, -s,** or **-t** flags is specified. If this flag is omitted, the output is formatted with 23 lines per page if the **-o** flag is omitted; otherwise with 65 lines per page. When the **-o** flag is given, a page eject is inserted at the beginning of each page.

**-b** Begin recording. If this argument is omitted, **ptxrlog** begins recording immediately. The flag and arguments are used to start the recording at a specified later time. The flag must be followed by the start time in the format *hhmm*, where:

*hh* = Hour in 24 hour time (midnight is 00).

*mm* = Minutes.

**-e** End recording. Specifies the number of hours and minutes recording must be active. The flag must be followed by the number of hours and minutes in the format *hh.mm*, where:

*hh* = Number of hours to record.

*mm* = Number of minutes to record.

If this argument is omitted, the recording continues for 12 hours. A maximum of 24 hours can be specified. When the time specified by this argument has elapsed, **ptxrlog** terminates.

## Binary Recording Files

When the **-r** flag is used, output is written to the file name specified after the flag. If the file exists when recording starts, it is opened for append. After opening the binary output file, whether for creation or append, **ptxrlog** writes the control records to the file. For existing files, this causes the file to contain more than one set of control records and may require you to process the file with **ptxmerge** or **ptxsplit** before you can process the file with **xmperf** or **azizo.**

## Resynchronizing with ptxrlog

The **ptxrlog** program initiates a resynchronizing with the data-supplier host if the data-supplier host sends an **i_am_back** packet. This usually happens if the data-supplier host's **xmservd** daemon has died and is restarted.

The **ptxrlog** program also initiates a resynchronizing with the data-supplier host if no **data_feed** packets have been received for ten times the specified sampling interval.

# Listing Recorded HotSet Data with ptxhottab

The **ptxhottab** program was included after support for HotSet data was added to PTX. The program is used to tabulate data from a recording file like **ptxtab**, but it tabulates HotSet data while ignoring other data.

## The ptxhottab Command Line

The **ptxhottab** command line looks as follows:

**ptxhottab** [**-c**] *recording_file*

**-c** Condensed output

The program has two output formats: uncondensed and condensed. Uncondensed output uses a format with name-equals-value pairs separated by semicolons.

# Processing HotSet Recordings with ptx2stat

The **ptx2stat** program was included after support for HotSet data was added to PTX. The program is used to modify the HotSet data collected in a recording file so it appears to be collected in StatSets. Because HotSets and StatSets are inherently different, and especially because HotSet data is likely to exist for only a few time periods, the conversion can usually not make the data appear exactly as StatSet data.

The erratic scattering of actual observations makes it difficult to see observations over a time period when played back with **xmperf**. To make it more usable, it is a good idea to postprocess the converted recording file so that the old HotSet data appears in the same StatSet as real StatSet data. The converted HotSet observations then appear as ″blips″ in the playback window.

For use with **azizo**, conversion with **ptx2stat** can be done so each observation is followed by a stop record.

## The ptx2stat Command Line

The **ptx2stat** command line looks as follows:

**ptx2stat** [**-s**] *infile outfile*

**-s** Insert stop-records after each value set.

# Chapter 10. Analyzing Performance Recordings with azizo

The **azizo** recording application has been replaced with the java-based **jazizo** application in Performance Toolbox for AIX Version 3. This new tool provides more functionality than **azizo** and is easier to use. The **jazizo** application processes long-term, large metric set recordings from the **xmtrend** daemon.

## Initial Processing of Recording Files

The **azizo** program analyzes one recording file at a time. If multiple recordings must be analyzed together, the support program **ptxmerge** can be used to merge multiple recording files into one for simultaneous analysis of statistics from multiple sources.

When recording files contain console definitions, the definitions are not usable in the analysis performed by **azizo** and are ignored, except when a filtered recording is produced. All other record types in recording files are used to build the data tables used in the analysis.

Whenever **azizo** reads a recording file, it first finds all the statistics defined in the file. For each statistic, it builds a table with a number of elements corresponding to the width of the graph area. Each element has the following fields:

**Maximum**
The highest observation received for this statistic in the time interval corresponding to the table element.

**Minimum**
The lowest observation received for this statistic in the time interval corresponding to the table element.

**Sum**
The sum of all observations received for this statistic in the time interval corresponding to the table element.

**Count**
The count of observations received for this statistic in the time interval corresponding to the table element.

The same statistic can occur in more than one of the sets of statistics (statsets) in the recording file and it can occur more than once in any set. All such statistics are collected into one table by **azizo**.

After building the statistics tables, **azizo** determines the time period covered by the recording. From this, it calculates how time stamps correspond to elements in the statistics tables. Next, all value records are read and their observations are routed to the table elements that correspond to the value record's time stamp.

When extracting statistic values from the recording file, **azizo** uses one of the two observation fields in the value record depending on the type of the statistic. Statistics can be of type **SiCounter** or of type **SiQuantity**:

**SiCounter**
The value of such statistics is incremented continuously by the monitored system. The delta value, which represents the difference between two consecutive observations, is used by **azizo** for this type of statistic. When moved to the tables, the observation is converted into a rate per second.

**SiQuantity**
Value represents a level, such as memory used or available disk space. The actual observation value is used by **azizo**.

When the entire recording file has been read, **azizo** calculates statistical data for each statistic. This includes:

**Average**
The average of all observations covered by the recording.

**Standard deviation**
The standard deviation.

© Copyright IBM Corp. 1994, 2004 

**Maximum** The highest value of all observations covered by the recording.

**Minimum** The lowest value of all observations covered by the recording.

In the next step, **azizo** creates small graphs for each of the statistics it has detected. Such graphs are called *metrics graphs* and are displayed in the **azizo** main window in a scrollable list.

The maximum number of statistics **azizo** can process is 256. If greater than 256 statistics exist in a recording file, metrics graphs are created for only the first 256. When greater than 256 statistics are encountered, **azizo** informs the user through a message window. Even though only 256 metrics graphs are created, if you use the **Rescan** option when zooming in on a main graph, all the statistics in the recording file are included in the scan because the scan is done on the actual file rather than the set of metrics graphs. Up to 256 metrics are included in the zoomed-in main graph but some of those may not have a corresponding metrics graph. Zooming-in on main graphs is described in “Zooming-in on Main Graphs” on page 119.

As the final step, **azizo** creates a main graph in a separate window. This graph contains all or a selection of the statistics in the recording file in a single graph, depending on the number of statistics at hand and the characteristics of the observations for each.

## The azizo Main Window

The **azizo** main window is divided horizontally into the following three sections:
- Icon section
- Metrics selection window
- Message window

The main window has full window decorations, but the **Close** option of the window manager menu has been disabled to prevent accidental exit of **azizo**.

### The Icon Section

At the top of the main window is the icon section. The icon section of the window always has a fixed height. It is subdivided into three sections:

**Actions** This section contains icons, which represent actions that can be applied to supported objects. Generally, the action is performed when a compatible object is dragged to an icon and dropped there.

**Files** This section contains icons representing locations where recording files can be retrieved from. To activate a source, select the corresponding icon to open a window with possible choices. Currently, this section contains only one source, represented by the Local Files icon.

**Zoom Views** Currently unused. This section is intended as a graphical illustration of the current zoomed-in views of the recording.

### The Metrics Selection Window

The metrics selection window occupies the middle part of the main window. It is the only section of the main window that changes height when the main window is resized. At its top is the title part, which identifies the recording file and shows a time scale and the time stamps of the earliest and the latest observation in that file. Following the title part comes a scrollable list of the metrics graphs that were created from the statistics contained in the recording file.

The title part is considered an object as is each of the metrics graphs. Actions can be performed on individual metrics by dragging their metrics graphs to an action icon. Actions can be performed on all of the metrics in the metrics selection window by dragging the title part object to an action icon.

“Using the azizo Metrics Selection Window” on page 114 describes how to work with the metrics selection window.

## The Message Window

The message window is a scrollable text window at the bottom of the main window. It is used to display messages from **azizo** to the end user. Approximately 80 message lines are retained for the user to scroll back in.

Error messages sent to the message window are also logged to the **azizo** log file in **$HOME/azizo.log**. This log file is overwritten each time **azizo** starts and is not closed until **azizo** terminates. Therefore, its contents are not dependable until you exit **azizo**.

## Main Graphs

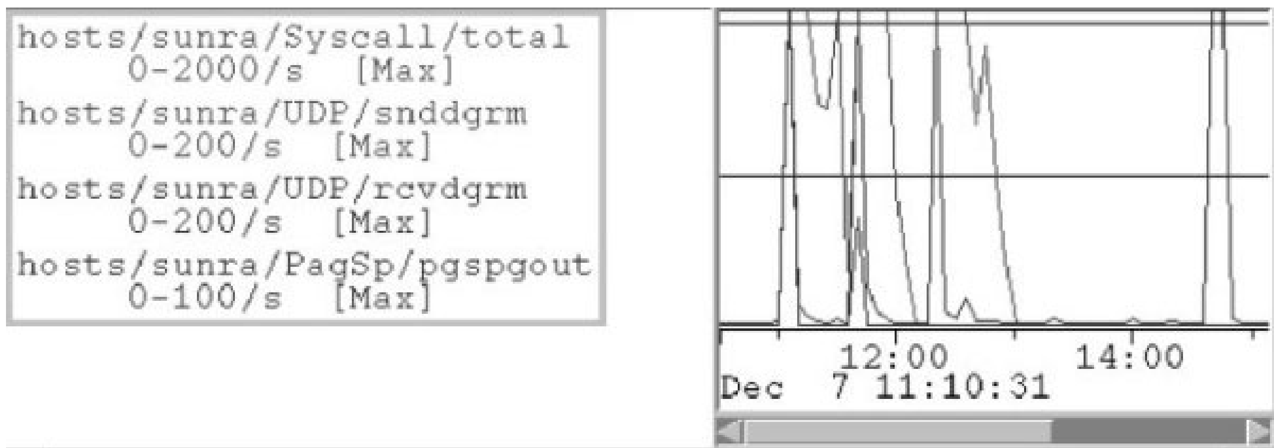



*Figure 5. Top-level Main Graph, azizo. This figure shows the scrolling list of metric names on the right and a scrolling histrogram graph of the metric values on the right.*

Main graphs are the principal viewing windows of **azizo**. When **azizo** reads a recording file, it always displays a top-level main graph that covers the entire time interval of the recording file. You can create additional main graphs by zooming in on the top-level graph or any zoomed-in main graph.

Main graphs contain two sections. To the left is a list of metrics that are included in the graph. The metric names are displayed using the same color as is used to draw the data. If the list of metrics is longer than the window can display, a scroll bar allows you to scroll the list of metrics.

To the right is the actual graphical display of the metrics data for the time period covered by the graph. Graphs are drawn from tables with elements corresponding to time intervals of equal length. Each element can house one, multiple, or no raw observations from the recording file. The drawing style is line graph, meaning that the observation points for a metric are connected by a line drawn in the assigned color. Where the recording file contains stop records to signify that recording for one or more sets of statistics was stopped, the line graphs are broken so that it is immediately apparent that some elements do not contain data.

Lines for a metric can be drawn from the maximum value encountered in the time intervals, from the minimum value, from the average value, or from both the minimum and the maximum values. Below the actual graph area is a scale that divides the displayed graph into seconds, minutes, hours, days, or weeks, depending on the time period covered by the graph. The scale carries time stamps that are inserted as close as space permits. The start time and the stop time limiting the interval covered by the graph are displayed at the ends of the scale.

If you resize the window containing a main graph, the graph adjusts to the changed height. It does not adjust to a changed window width, except when the window is made so narrow that the whole graph cannot be displayed. In that case, a scroll bar is displayed so you can scroll the main graph horizontally.

### Top-Level Main Graph

When the top-level main graph is created, metrics are ordered after a factor calculated as the maximum value encountered in the recording divided by the anticipated maximum value for the metric as defined in the recording file. Included in the top-level main graph is up to a user-defined number of metrics. This number is defined by the X resource **MainGraphCount**. It defaults to 16. The metrics to be included are selected from the top of the ordered list.

Colors are allocated from the colors defined by the X resources **ValueColor1** through **ValueColor24**. If greater than 24 metrics are displayed, the color table is reused.

The width of the top-level graph is determined by the X resource **MetricWidth**. It defaults to 600 pixels. This effectively sets the width of the top-level main graph to the same as that of the metrics graphs in the metrics selection window and allows for easy correlation between the two windows. The height of the top-level graph is set according to the X resource **MainGraphHeight**, which defaults to 300.

A top-level main graph cannot be deleted. However, you can remove all the metrics from it in one operation.

### Zoomed-in Main Graph

A zoomed-in main graph is always derived from some other main graph. Its ancestor may be the top-level main graph or a zoomed-in main graph. The zoom-in operation is described in “Zooming-in on Main Graphs” on page 119.

All zoomed-in main graphs are created with a width determined by the X resource **MainGraphWidth** and a height set by the X resource **MainGraphHeight**. These two default to 600 and 300 pixels, respectively. Zoomed-in main graphs can be deleted by dragging them to the **Pit** icon. If a main graph has zoomed-in descendants, all those are deleted when the main graph itself is dragged to the **Pit** icon.

## The azizo Command Line

The **azizo** program is started with the following command line:

**azizo** [**-f** *recording_file*]

The command line argument is optional and has the following meaning:

**-f**
Recording file path name. Used to specify the name of a recording file to analyze. If the file is a valid recording file, **azizo** reads the file and processes it. If this argument is omitted or the specified file is invalid, **azizo** starts and awaits your selection of a recording file by selecting the **Local Files** icon.

Beginning with Version 2.2, this argument also brings up the recording’s existing annotation file, or creates a new one, so that the user can take notes about the recording.

## The azizo User Interface

This section explains the mechanics of the **azizo** user interface. It is significantly different from the user interface of the **xmperf** program because it uses a drag-and-drop technique rather than a menu interface.

The drag-and-drop interface is based on a notion of objects, which are things you can manipulate, and actions, that represent the ways you can manipulate the objects. In most cases, you specify what you want by dragging an object to an action. If the action is valid for the object, the action is performed on the object, sometimes after further interaction with you.

Most actions are represented by icons but main graph windows also represent actions for certain types of objects. For example, you can add a metric to a main graph by dragging a metrics graph object to the main graph.

A few operations are not implemented as drag-and-drop operations. They are explained in the sections, “Non-drag Operations” on page 113 and “Zooming-in on Main Graphs” on page 119.

## Icons

Icons are the little square buttons displayed in the Actions and Files sections of the **azizo** main window. The icons in the Files section act as activators for windows that you use to access recording files. Select one of these icons to activate the corresponding window.

Icons in the Actions window are mostly receivers of objects on which you wish to perform the associated action. For example, if you want to delete an object, you drag it to the **Pit** icon and drop it there. When you drag an object to an icon, you can immediately see if the action represented by the icon can be performed on the object. If it can, the icon changes shape to indicate that it accepts objects of the type you drag. If the icon does not accept objects of this type, the icon remains unchanged and an attempt to drop the object causes the dragged object representation to snap back to the object.

All icons, with the exception of the **Help** icon itself, can be dragged to and dropped onto the **Help**. This opens a help window that explains the function of the dragged icon.

Usually, icons are colorful little things that are generated from icon description files in directory **/usr/lib/X11/app-defaults/perfmgr/icons** on your system. If one or more icon description files have been deleted, or if resources (such as colors) are not available on your system, the icons are displayed as text buttons. Whenever this occurs, the **azizo** log file in your home directory (**$HOME/azizo.log**) explains the reason for not displaying color icons.

The text shows one of the following error codes:

```
 1 Error matching color
-1 Unable to open icon description file
-2 Icon description file has invalid data
-3 X Server is out of memory
-4 Unable to allocate color
```

**Note:** The log file contents are not dependable until you exit **azizo**.

Icon description files must be in the format defined by XPM2 (X Pixmap Format 2).

## Dragging and Dropping

Dragging is performed with the mouse. To drag an object, move the pointer over the object you want to drag, then press the right mouse button. Keep the mouse button pressed and move the pointer to where you want to drop the object.

As you move the pointer, the shape of the pointer changes to a drag icon that identifies the object you are dragging. At the top left of the drag icon is a small arrow. The tip of the arrow is considered the active point of the drag icon.

If the action you drag the object to can be performed on the object, the action indicates this by changing shape. Where the action is represented by an icon, the icon itself changes to a slightly larger icon. If the

action is represented by a window, the window is surrounded by a highlighting frame. This allows you to validate a drag-and-drop operation before you complete it.

To complete the drag-and-drop operation, release the right mouse button while the active point of the drag icon is within the outline of the action. If the action can be performed on the object, the drag icon displays to be absorbed by the action icon or window. If the action can't be performed on the object, the drag icon is displayed to "snap back" into the object as if connected to it with a rubber band.

A special action is represented by the help icon in the action icon section of the main window. You can drag other action icons and objects to this icon to display a help screen that explains the action or object. For each action, the help screen explains to which objects it can be applied; for each object, the help screen explains which actions can be performed on it.

## Objects

The **azizo** program defines the following objects:

**Main Graph**
The actual graph part of a main graph window. The object is considered to be the main graph as a whole, including the metric labels in the left part of the main graph window.

**Metric Label**
One of the metric labels in the left part of a main graph window. The object is considered to be the metric as it is displayed in the main graph.

**Title Part of Metrics Selection Window**
The title part of the metrics selection window. The object is considered to be all of the metrics or metrics graphs currently part of the metrics selection window.

**Metrics Graph**
The object is the metric identified by the metrics graph or the metrics graph itself.

**Information Window**
The object is the entire information window.

**Config Icon**
The object is the configuration file.

**Configuration Line**
The object is a single named configuration from the configuration file.

Each of the defined objects is associated with a different drag icon. If your current setup prevents the drag icon from being created as a color drag icon, a simpler icon that shows the name of the drag icon is used.

## Actions

Most of the actions you can perform on **azizo** objects are represented by icons. The only exception is that main graphs can act as actions for certain objects. The full list of actions is the following:

**Config Icon**
The action is to save the current view of a main graph to the configuration file.

**Info Icon**
The action is to display an information window with a tabulated view of the statistical data for a single or multiple metrics.

**Print Icon**
The action is to print a graphical or tabulated view of metrics, optionally to a file.

**Filter Icon**
The action is to produce a filtered recording file, based upon a main graph.

**View Icon**
The action is to change the way a graph is drawn.

**Scale Icon**
The action is to change the scale used to draw a main graph.

**Annotate Icon**
Available with Version 2.2 or later, the action of this icon is to open an editor window for creation or modification of annotations.

**Pit Icon** The action is to remove an object.

**Help Icon** The action is to display a help screen for an object or action.

**Main Graph** The action is to add a metric (if the drag object is a metrics graph) or to apply a presaved view from the configuration file to the graph.

## Non-drag Operations

Non-drag operations are defined for special cases. Non-drag operations on main graphs are described in “Zooming-in on Main Graphs” on page 119; others in the sections below.

## Selecting a Recording File

When you want to select a recording file to analyze, this can be done on the command line when you start **azizo** or it can be done by selecting the **Local Files** icon in the Files icon section of the main window. When you select the icon, the Select Recording File selection box opens.

It is a standard file selection box, which allows you to change the filter to show any selection of files that match the filter. The default filter is:

```
$HOME/XmRec/R.*
```

All **xmperf** and **3dmon** recordings in the designated recording directory for your user ID match this filter. To see all **xmservd** recordings on your machine, change the filter to the following and select **Filter**:

```
/etc/perf/azizo.*
```

When the file you wish to analyze is shown in the Files section of the selection box, double-click on the file name, or select the file name and then select **OK**. This causes the selected file to become the current recording file. All previous main graphs are removed and so are all metrics graphs in the metrics selection window; **azizo** then resets itself and reads the new file as described in “Initial Processing of Recording Files” on page 107. When a top-level main graph is displayed, you can start working with the new file.

## Exiting azizo

To exit the **azizo** program, you must select the **Exit** icon with the left mouse button. This opens the Exit azizo dialog box which asks you to confirm that you want to exit the program by selecting **OK**. To resume using the program, select **Cancel**.

## The Help Facility

In **azizo**, help can be invoked in three ways:

- You can select the **Help** icon to be taken to the index of help topics.
- You can drag an object or an action icon to the **Help** icon and drop it there to get help on the object or action.
- You can select the help button in a dialog window to get help on how to use the dialog box.

The help function depends on the presence of a help file for **azizo** with a format as described in “Simple Help File Format” on page 286. The file is looked for under the name **$HOME/azizo.log** in the user’s home directory. If the file is not found there, it is searched for as described in Appendix B, “Performance Toolbox for AIX Files,” on page 271.

The help file contains help texts that are identified by a string of characters. Every object and every action defined by **azizo** and every dialog box has an identifier (a string of characters) associated with it. Whenever you select help by dragging an object or action to the **Help** icon or from a dialog window, the identifier of the object, action, or dialog box is used to locate the associated help text. In the resulting help window, the window frame shows the identifier used to locate the help text.

## Browsing

You can browse through the help text by using the scroll bar to the right in the help window. The help window shows up to 25 lines at a time.

## Help Index

From any help window, you can get to an index of help topics by using the pull-down Help menu and then select **Help Index**. The index is a list of all help identifiers available. When you select a line, a help window is created with the associated help text. From the menu bar's **Help** item, you can select the menu item **Help on Help** for detailed instructions on how to use help.

## Annotating a Recording

When using Performance Toolbox for AIX, Version 2.2 or later, to create or modify an annotation to the current recording file, drag the graph area of the main graph to the **Annotate** icon and drop it there. An editor window will open. From the editor window, you can modify any existing annotation text or create a new annotation file if none existed before.

When you have made the changes you want, exit the editor window with the **file** or **save** option of the editor in use. If you exit the editor with the **quit without saving** option of the editor in use, any existing annotation file is left unchanged and no new annotation file is created if none existed before.

# Using the azizo Metrics Selection Window

The metrics selection window is created as an OSF/Motif widget of type **XmForm**, which contains an **XmLabel** widget to display the title part of the window and an **XmScrolledWindow** widget used to display a scrollable list of metrics graphs. If all the contained metrics graphs can be displayed in the window, the vertical scroll bar is missing; if the full width of the metrics graphs can be shown in the window, no horizontal scroll bar is present. When the vertical scroll bar is active, it can be used to scroll through the metrics graphs. When the horizontal scroll bar is visible, it can be used to scroll the entire set of metrics graphs left and right.

## The Title Part of the Metrics Selection Window

The title part of the window has a left part that gives information about the way metrics graphs are displayed and a right part, which shows the time scale and other information as it applies to all the metrics graphs.

The left side of the title part shows the graph style used to draw the metrics graphs. If the same style is not used for all metrics graphs, the graph style is shown as Mixed. Below the graph style is a line showing the meaning of the tick marks of the scale in the right section of the title part. It may show that each tick mark on the scale corresponds to a minute, an hour, a day, or a week, depending on the time period covered by the graph.

The right side of the title part contains a time scale that applies to all metrics graphs. It is aligned so each tick mark on the scale corresponds to full minutes, hours, days, or weeks. Above and at each end of the scale is a full time stamp that shows the times of the earliest and the latest observation as encountered in the recording file. Above and over the center of the scale is shown the file name of the current recording file. If the title part is too narrow to allow time stamps and file name to be displayed side by side, the file name is overlapping the time stamps.

The end time is determined by the highest time stamp found among the last several value records of the file. If the current recording file is a file produced by concatenating multiple recording files into one without using the **ptxmerge** program, then you may see interesting, though not necessarily useful, side effects, especially when the end time is lower than the start time. It is not recommended to use **azizo** to analyze such files without first using the **ptxmerge** program to rearrange records in the files or using the **ptxsplit** program to split the file into its individual recordings.

## Metrics Graphs

Each metrics graph has a left side used to display the full path name of the metric it represents and a right side that displays a graph representing the observations for the metric. The width of the left side is set by the X resource **MetricLegendWidth**. When the graph is initially created, the style of the graph is determined by the horizontal scale of the graph and the X resources **MetricLinePlot** and **MetricLineDouble**. The colors used to draw the graph are determined by the X resources **OnlyMetricColorIndex**, **FirstMetricColorIndex**, and **SecondMetricColorIndex**. The width of the individual metrics graphs is set through the X Resource **MetricWidth**.

By default, neither the **MetricLinePlot** nor the **MetricLineDouble** resource is set `true`. The style then defaults to `Bar, Max-to-Min`. If each pixel on the horizontal scale corresponds less than one second, the `Bar, Max-to-Min` style is not meaningful and the default style is changed to `Line, Max & Min`.

The style of individual or all metrics graphs can be changed as explained in “Changing the Style of Metrics” on page 117.

## X Resources for the Metrics Selection Window

The following X resources control the initial appearance of metrics selection windows:

**AZMetrics**
The name of all widgets used to create the metrics selection window. By referencing the **AZMetrics** widget name, you can set the foreground and background color of the metrics selection window and the metrics graphs. The colors you assign for foreground becomes the default color for drawing the metrics graphs. It can be overridden by other X resources as explained below. To set the background color of the metrics selection window, use ***AZMetrics.background** as the resource name.

**MetricLegendWidth**
Sets the width of the area at the left side of a metrics graph that is used to display the metric path name. Default is 20 characters; permitted range is 8 - 32 characters.

**MetricLineDouble**
This resource can be set to either true or false. It is ignored if the resource **MetricLinePlot** is set to false. If both these resources are true, the default way of drawing individual metrics graphs is with two lines: one for the maximum and one for the minimum values. If this resource is set false while **MetricLinePlot** is true, only the maximum values are drawn.

**MetricLinePlot**
This resource can be set to either true or false. If this resource and the resource **MetricLineDouble** are both set to true, the default way of drawing individual metrics graphs is with two lines: one for the maximum and one for the minimum values. If this resource is set to true while **MetricLineDouble** is false, only the maximum values are drawn. If this resource is set to false, individual metrics graphs are drawn in the `Bar, Max-to-Min` style as a series of vertical lines, each one connecting the maximum and minimum value in an interval.

**OnlyMetricColorIndex**
This resource is used to select the foreground color of a metrics graph when only one line is drawn in each metrics graph. The resource defines an index into a table of defined **ValueColor1** through **ValueColor24** resources to use for main graphs. Default is the foreground color of the metrics selection window as set by the **AZMetrics** resource.

**FirstMetricColorIndex**
This resource specifies the index into a table of defined **ValueColor1** through **ValueColor24** resources. The color selected by the index is used to draw the maximum value line of metrics in the metrics selection window when both maximum and minimum values are drawn. Default is foreground color of the metrics selection window as set by the **AZMetrics** resource.

**SecondMetricColorIndex** This resource specifies the index into a table of defined **ValueColor1** through **ValueColor24** resources. The color selected by the index is used to draw the minimum value line of metrics in the metrics selection window when both maximum and minimum values are drawn. Default is foreground color of the metrics selection window as set by the **AZMetrics** resource.

**MetricHeight** Sets the height in pixels of individual metrics graphs in the metrics selection window. Default is 40 pixels; permitted range is 10 - 200 pixels.

**MetricWidth** This resource sets the width of the individual metrics graphs in the metrics selection window. It also sets the width of the graph area of the top-level main graph. The default is 600 pixels. Specify in the range 100 through 1,000.

## Tabular View of Metrics

Each of the metrics graphs represents a single statistic as recorded in the current file. You can see a tabular view of the statistical data for one or all metrics in the metrics selection window by dragging a single metrics graph or the title part of the metrics selection window to the **Info** icon. The tabular view is shown in an information window similar to the following example. The color of information windows can be set through the X resource **InfoWindow**.

```
Summary for graph 100%, Jul 28 00:00:05 1999 - Jul 28 23:59:59 1999
No. of  Average    Standard   Maximum   Minimum
Observ    Value   Deviation     Value     Value    Metric path name
------  -------   ---------   -------   -------    ----------------------
17280       877       14137    268505        28    sunra/SysIO/writech
17280       877       14137    268505        28    sunra/CPU/cpu0/writech
17280       580        5651    249235        48    sunra/SysIO/readch
17280       580        5651    249235        48    sunra/CPU/cpu0/readch
```

If the object you drag to the **Info** icon is a single metrics graph, a small window displays the statistical data as calculated for the corresponding metric. This window cannot be used as an object. All you can do is look at it, and then select **OK** to close the window.

If the object you drag is the title part of the metrics selection window, a larger information window opens. This window contains a line for each of the metrics graphs in the metrics selection window.

The tabular view can be printed by dragging the information window to the **Print** icon. To drag the window, place the mouse pointer anywhere within the data part of the window and start the drag operation from there. To close the window, select **OK**.

## Printing Metrics from the Metrics Selection Window

The metrics graphs can be printed individually or they can be printed as a selection of metrics graphs. To print an individual metrics graph, drag the metrics graph to the **Print** icon. To print multiple metrics graphs, position the vertical scroll bar of the metrics selection window so the metrics graphs you want to print are visible in the metrics selection window and then drag the title part of the metrics selection window to the **Print** icon. The printed image contains everything that is visible in the metrics selection window, including the title part. Tabular views of all metrics in the metrics selection window can be printed by dragging the data part of the information window to the **Print** icon.

After you drop a graph object on the **Print** icon to print one or more graphs, the Print Box dialog box opens. From this dialog box, you can specify the destination as a printer or a named file. You can also specify other options to use for printing. This is explained in “The Print Box” on page 126.

For printing of information windows, the Report Box opens. This dialog box allows you to specify the destination of the print output as a printer or a named file. You can also specify other options to use for printing. It is explained in “The Report Box” on page 127.

## Changing the Style of Metrics

The style of one or all metrics graphs can be changed by dragging the metrics graph or the title part of the metrics selection window to the **View** icon. This causes the Changing View Options dialog box to pop open. From this dialog box you can change the graph style of the metrics graph or all the metrics graphs into either of:

- Line, Maximum
- Line, Max & Min
- Bar, Max-to-Min

These styles and the dialog box are explained in “Changing View Options Dialog Box” on page 129.

## Removing Metrics

If a metrics graph represents a statistic you don’t need in the analysis, you can remove it from the metrics selection window. You do so by dragging the metrics graph to the **Pit** icon. When the metrics graph is removed, the corresponding metric is also removed from all the main graphs it is part of.

When a metric is removed from the metrics selection window, it cannot be added back to the metrics selection window except by rereading the recording file. However, if you use the **Rescan** option when zooming in on a main graph, the deleted metric is included in the scan, because the scan is done on the actual file rather than the metrics selection window.

# Working with azizo Main Graphs

Main graph windows are created as OSF/Motif top-level windows of the **ApplicationShell** widget class. They contain two subwindows, both of which are scrollable. To the left is the metrics label part, to the right is the actual graph part of the window. The parts have horizontal scroll bars if the current width of the main graph window doesn’t allow all of the information in the windows to be displayed in its entirety.

The metrics label part have a vertical scroll bar if the height of the main graph window doesn’t allow all metric labels to be displayed at the same time. The graph part never has a vertical scroll bar. It resizes the height of the graph area to fit the size of the enclosing main graph window. The graph area might not be resized after the main graph window is made smaller. Requesting a refresh from the window manager forces the resizing.

## The Main Graph Window Frame

The window frames of main graph windows have full window manager decorations. However, the **Close** option of the Window Manager menu is disabled to prevent accidental closure of a main graph window.

The title bar of a main graph window indicates the degree of zoom-in applied to create the window. The top-level main graph always has the text *100%* in the title bar. All other main graphs have a title bar that shows the time span they cover in percent of the time span of the top-level main graph. Such percentages are shown with two decimal places so that a graph derived from the top-level graph but covering the same time span, has a title bar that says *100.00%*.

Because it is possible to have multiple derived main graphs that cover the same percentage of the total time or even the exact same interval, multiple main graph windows can have identical title bars.

## The Metric Label Part of a Main Graph Window

Metrics represent statistics and are identified by the full path name of the statistic. The metrics label part of a main graph window contains a list of path names to illustrate which metrics are included in the main graph. Each entry in the list is called a metric label.

Each of the metric labels consists of two lines. The first line shows the full path name of the metric. The second line first lists the scale used when drawing the line graph of the metric. The scale is given as a from-to value. Note that the scale for a statistic of type **SiCounter** represents a rate per second.

The metric label part of the main graph window has a fixed width determined by the longest path name displayed and the font in use. Occasionally, the metric label part of the main graph window is too narrow to show the entire path name. When this happens, a horizontal scroll bar allows you to scroll horizontally in the entire list of metric labels.

Following the scale and on the same line, is the current drawing style for the metric. The style is enclosed in square brackets and is one of the following words or abbreviations:

- Avg
- Max
- Min
- Both

The meaning of these and how to change the style of a metric is explained in the sections "Changing the Appearance of Main Graphs" on page 121 and "Changing View Options Dialog Box" on page 129.

Colors are assigned to metrics in the order they are displayed in the metrics label part. The first metric gets the color defined by the X resource **ValueColor1**, the next gets the color defined by **ValueColor2**. This continues until there are no more metric labels to add or until the 24th color is assigned. If more labels need to be assigned a color, assignment starts over again from **ValueColor1**.

## The Graph Part of a Main Graph Window

The graph part of the main graph window is used to draw a line graph for each of the metric labels in the left part of the window. The lines are drawn in the same colors as are used to display the metric labels.

The drawing area extends from the top inner frame of the enclosing window and down to the scale line at the bottom of the window. The distance between the scale line and the bottom inner frame of the window is twice the height of the active font.

The graph has four horizontal lines drawn to indicate the 25%, 50%, 75%, and 100% vertical scale lines. Only the 50% and the 100% lines are drawn if the height of the graph area is too small to make four vertical scale lines feasible. The 100% vertical scale line is positioned so it is possible to draw values up to approximately 105% within the drawing area. As an illustration of the use of vertical scale lines, assume a metric label shows a scale of 0-2000. The 25% vertical scale line for this metric would correspond to an observation value of 500.

The line graphs are drawn according to the style selected for the metrics as shown in the metric labels. Initially, all metrics have the same style, as determined by the X resource **MainPlotStyle**. Styles for individual or all metrics can be changed later as described in "Changing the Appearance of Main Graphs" on page 121 and "Changing View Options Dialog Box" on page 129.

The scale line at the bottom of the graph area has tick marks that divide the interval covered by the main graph into units of seconds, minutes, hours, days, or weeks. Some of the tick marks are slightly longer than others and correspond to time stamps placed directly below.

At the bottom left of the graph area is a time stamp that gives the time of the earliest observation included in the graph. At the bottom right of the graph area is a time stamp that gives the time of the latest observation included in the graph.

## X Resources for Main Graphs

The following X resources control the initial appearance of Main Graphs:

**GraphFont**
The font to use for all text in **azizo**. Defaults to a fixed-width font suitable for **azizo**. The font determines how much space is reserved for scale and time stamps at the bottom of the graph area and how closely time stamps can be spaced under the bottom scale line.

**GraphWindowWidth**
The initial width of main graph windows. This width is the width of the window itself, not the main graph area alone. Default is 862 pixels.

**MainGraphCount**
Defines the maximum number of metrics that are part of the top-level main graph after a new recording file has been read in. Defaults to 16 metrics.

**MainGraphHeight**
Sets the initial height (in pixels) of all main graphs. Default is 300 pixels; permitted range is 50 - 1,000 pixels.

**MainGraphWidth**
Sets the initial width (in pixels) of all main graphs except the top-level main graph. Default is 600 pixels; permitted range is 100 - 1,200 pixels. Note that main graphs are never scaled horizontally. If you increase the width of the main graph window beyond what is required to show the full graph width, empty space is displayed to the right of the graph. If you reduce the width of the main graph window to less than what is required to display the main graph, then a scroll bar is added to allow horizontal scrolling of the graph area.

**MainPlotStyle**
Sets the default plotting style for metrics in main graphs. Defaults to average. Permitted values are: maximum, minimum, both, and average.

**MetricWidth**
Sets the width of the graph area of the top-level main graph. Zoomed-in main graphs get a width as set by the **MainGraphWidth** X resource. The default is 600 pixels. Specify in the range 100 through 1,000.

**MonoLegends**
If this resource is set true, all labels in main graphs are shown in the foreground color of the label part of the main graph. It is probably not useful because then it is impossible to see which line in the graph corresponds to the label.

**MainGraph**
The name of all the widgets used to create main graphs is **MainGraph**. By assigning color values to the X resources defined as **MainGraph*XmLabel.background** and **MainGraph*XmLabel.foreground**, the appearance of the metrics label part can be changed. Similarly, the colors used in the graph part can be changed with the resources **MainGraph*XmDrawingArea.background** and **MainGraph*XmDrawingArea.foreground**. The supplied X resource file for **azizo** sets the background color for both parts of the main graph window.

## Zooming-in on Main Graphs

One of the vital functions of **azizo** is the ability to zoom-in on subsections of a main graph. The zoom-in is achieved by drawing an outline in the main graph area and then selecting the type of zoom-in from a dialog window. The outline is shaped as a rectangle and defines an area of interest.

You draw the outline by moving the mouse pointer to the place where you want one of the corners of the rectangle to be. Then press the left mouse button and hold it down while moving the mouse pointer. This draws a rubberband outline of the rectangle. When the outline has the size you want, release the mouse

button. The left and right sides of the rectangle define the area of interest in time. The top and bottom sides of the rectangle declare the observation value area of interest.

When the outline is drawn, the Zoom-in dialog box opens. From this dialog box, you must select **Rescan** or **Keep Metrics** to go through with the zoom-in operation.

## Rescan

When you select **Rescan**, the left and right sides of the rectangle are used to exclude all observations in the recording file that fall outside the time period limited by the two sides. Then all remaining observations are analyzed to see if any one of them has at least one observation that falls within the top and bottom lines of the rectangle. When determining if an observation falls within the area, the plot point is calculated from the assumption that the 100% line in the graph area corresponds to the anticipated maximum value as retrieved from the recording file. All metrics that have at least one observation within the area of interest are included in the zoomed-in graph.

Note that a metric can have an observation value that falls within the outline without the source graph showing this. It often is the case when each interval contains multiple observation values because the main graph shows only the maximum, minimum, or average values for each interval.

## Keep Metrics

When you select **Keep Metrics** the zoomed-in graph is created with the same metrics as the source main graph. The left and right sides of the rectangle are used to exclude all metric observations in the recording file that fall outside the time period limited by the two sides. This selection corresponds to a zoom-in in time.

### Adding Metrics to Main Graphs

For recording files that contain many statistics, it can be difficult to select exactly the metrics of interest through the **Rescan** option of the zoom-in operation. This can be remedied by adding and deleting single metrics from a main graph. This section describes how to add metrics to a main graph.

The operation is as simple as dragging a metrics graph from the metrics selection window to the graph area of the main graph window and dropping it there. If the metric is already part of the main graph, nothing happens. Otherwise, the metric is added to the main graph and both parts of the main graph window are updated to reflect this. The added metric is given the first free color from the table defined by the X resources **ValueColor1** through **ValueColor24**.

### Removing Metrics from Main Graphs

A metric can be removed from a main graph as easily as it can be added to it. To remove a metric from a main graph, drag the corresponding metric label from the metrics label area to the **Pit** icon. This removes the metric from the main graph and updates both parts of the main graph window to reflect the change.

## Removing Main Graphs

When a main graph is no longer needed, it can be deleted by dragging it to the **Pit** icon and dropping it there. This causes the main graph and all its descendants (all main graphs created by zooming-in on the dragged graph) to be deleted. If the dragged graph is the top-level main graph, this graph itself is not deleted but all its metrics are removed from it. To make the empty main graph show metrics again, you must drag individual metrics graphs to the main graph from the metrics selection window.

## Tabular View of Main Graphs

Each of the metrics in a main graph represents a single metric as recorded in the current file. You can see a tabular view of the statistical data as calculated for the metrics that are included in the main graph by dragging the graph area of the main graph to the **Info** icon.

This creates an information window that contains a line for each of the metrics in the main graph. If the graph is the top-level main graph, the information window has only one part, showing the statistical data as calculated for the entire time period covered by the recording file. If the graph is a zoomed-in main graph, the window has one part that displays the statistical data for the full time period and another showing them for the zoomed-in time period.

If a metric is part of a main graph but is no longer in the metrics selection window, statistical information for the metric in the full time period of the recording is not available and cannot be shown in the window. If the information window is printed, such metrics are not displayed in the part of the report that covers the entire recording time period.

The tabular view can be printed by dragging the information window to the **Print** icon. To drag the window, place the mouse pointer anywhere within the data part of the window and start the drag operation from there. To close the Information window, select **OK**.

You can also drag a metric label to the **Info** icon. This produces a small window that displays the statistical data as calculated for the corresponding metric. This window cannot be used as an object. All you can do is look at it, and then select **OK** to remove the window.

## Printing Main Graphs

If you drag the graph area of a main graph to the Print icon and drop it there, you can print the graphical representation of the entire main graph window. This is done from the Print Box dialog that opens after you drop the main graph. From the Print Box you must select the options to use when printing the graph and enter the description you want printed with the graph. You can select to print directly to one of the printers known by your system, or you can send the print image to a file. Print images are in PostScript format. The Print Box is described in “The Print Box” on page 126.

## Changing the Appearance of Main Graphs

The appearance of a main graph can be altered by changing the height of the main graph with the window manager. This causes the graph to be redrawn so the graph area fills the entire height of the enclosing window. Another way to change the appearance is to apply a previously saved configuration view to it. This is explained in “Using the azizo Configuration File” on page 123.

Finally, the appearance can be changed through either or both of the two drag-and-drop operations described below. You invoke these operations for a single metric in a main graph by dragging its metric label to the **View** icon or the **Scale** icon. You invoke them for all metrics in the main graph by dragging the graph area of a main graph to the **View** icon or the **Scale** icon.

## Dragging to the View Icon

Dragging a main graph to the **View** icon signals to **azizo** that you want to change the view (drawing style) for all of the metrics that are currently included in the main graph. When you drop the main graph on the **View** icon, **azizo** displays a dialog box titled Changing View Options. This dialog box allows you to change the way all metrics are plotted in the main graph. The view style you select becomes the default view style for the main graph.

Dragging a metric label of a main graph to the View icon tells **azizo** that you want to change the view (drawing style) of the corresponding metric in the main graph. The Changing View Options dialog box is displayed for you to select the new drawing style.

In either case, the view (drawing style) can be selected as one of the following:

- **Maximum**
- **Minimum**
- **Both**

- **Average**

The styles and the Changing View Options dialog box are described in section “Changing View Options Dialog Box” on page 129.

## Dragging to the Scale Icon

When you drop a main graph on the **Scale** icon, you see a dialog box titled Rescaling from which you can change the scale of all the metrics in the main graph. The same dialog box opens when you drag a metric label of a main graph to the **Scale** icon.

The Rescaling dialog box identifies the main graph by its title and if the object was a metric label, identifies the metric label by its path name. Where the main graph is the object, any change you select from this dialog box applies to all the metrics in the graph; otherwise only the selected metric is affected.

The dialog box gives you three options for setting the scale of the metric:

**Autoscale**
The maximum value for the metric encountered in the interval covered by the main graph is used to determine the scale used to plot the metric. The scale is chosen so the maximum value falls between the 50% and the 100% lines in the graph.

**Normscale**
The scale is reset to its initial value as determined by the metric information carried in the recording file.

**Cancel**
Leave the scaling unchanged.

When one of the first two options is selected, both parts of the main graph are updated to reflect the change.

## Filtered Recordings

In most instances, a main graph represents a subset of the observations available in a recording file. The subset may be defined by a reduced time interval, by a reduced number of metrics, or both. When it is desired to preserve such a subset for later analysis, as a source for creating merged files, or for other reasons, you can use the filtering function of **azizo** to produce a filtered recording file. Filtering is initiated by dragging the graph area of a main graph to the **Filter** icon and dropping it there.

Filtering, thus, is the writing of a subset of the current recording file to a new file, the subset being defined by the set of metrics that are part of the main graph, which you dragged to the filter icon. For example, if the current recording file contains 20 metrics but the main graph shows only five of those, the filtered file contains data values for only five metrics.

Even though the filtered file contains data values for only the metrics that are part of the main graph, the filtered recording file contains all metrics definitions from the original file. By default, it also contains the other control record types that allow the filtered file to be played back by the **xmperf** program. The default can be changed as described in “Maintaining Instrument Definitions when Filtering” on page 123.

After you drop the main graph on the **Filter** icon, the Writing Filtered Recording dialog box opens. This dialog box has default values for the lowest and the highest time stamps to be included when data values are copied to the filtered output file. The defaults correspond to the interval covered by the main graph. You can change either time stamp to extend or reduce the time period covered by the filtered output, as long as the time period covers at least 2 seconds.

If you specify a start time lower than that of the oldest data value in the source recording file, copying starts from the beginning of the source file. If you specify an end time higher than that of the youngest data value in the source recording file, copying continues to the end of the source file.

The default file name for the filtered output is the source file name with **.filt** appended. You can change the file name to anything you want. If the file exists, you are asked whether you want to overwrite it.

## Maintaining Instrument Definitions when Filtering

If the source recording file was produced by **xmperf**, it includes data that defines the console and instruments from which the recording was produced. Recordings produced by other programs never have console information but always have a grouping of statistics into sets, which would be interpreted as instruments when the file is played back by **xmperf**.

A single button in the Writing Filtered Recording dialog box allows you to specify whether you want any console and statset definitions in the source recording file to be carried over to the filtered output file. This button is only displayed if the source file contains more than one set of statistics (statsets) or contains a console definition. For source files produced by **xmperf**, neither console nor instrument information is retained if you elect to not maintain instrument definitions. For source files produced by other programs, the grouping of statistics into sets is broken. In all cases where you elected not to maintain instrument definitions, the filtered recording contains only one set of statistics.

## Handling of Annotation Files when Filtering

**Note:** Annotation files are available with Version 2.2 or later only.

As described in "Annotation Files" on page 95, only recording files with names beginning with **R.** can have annotation files associated with them. Consequently, the azizo program only considers copying and merging of annotation files if the file created by the filtering function follows the recording file naming conventions.

If this is the case, an annotation file is created for the recording file produced by filtering if the original recording file had an associated annotation file or if an annotation file already exists that matches the recording file produced by the filtering.

The new annotation file is created using these steps:

1. Copy any existing annotation file that matches the name of the recording file produced by the filtering operation to a temporary file, surrounding the text with lines that identify the source of the text.
2. Append to the temporary file any annotation text that exists for the source recording file, surrounding the text with lines that identify the source of the text.
3. Rename the temporary file to match the recording file name produced by the filtering.

After the new annotation file is created, an editor window is opened for you to make any changes to the resulting annotation file.

## Using the azizo Configuration File

The **azizo** configuration file resides in each user's home directory. The file name is **$HOME/azizo.cf**. The file contains user-defined configurations that provide standardized views to be applied to main graphs.

Configurations are named entities that save a customized view of a main graph. Configuration views are used for viewing different recordings in a uniform manner that allows you to immediately compare the data in the recordings.

The configuration file is a binary file that cannot be modified by the end user, except through the interface provided by **azizo**.

### Saving Configurations

The current view of a main graph can be saved by dragging the graph part of the main graph window to the **Config** icon. This drag-and-drop operation indicates to **azizo** that you want to save the view of the

main graph as a named configuration that can later be recalled and applied to other main graphs. After you drop the drag icon, the Writing Configuration dialog box shown as follows.

For the configuration to be saved, you must enter a name for it in the dialog box. If the name you specify already exists, you are given the option of overwriting the existing configuration or changing the name of the new one you want to create.

In the dialog box, you can specify which parts of the combined view of the dragged main graph you want to be part of the saved configuration. You can elect to include or exclude the following options:

**Individual style**
Individual style is the view option you may have specified for one or more of the metrics in the main graph. When this selection is active, metrics in any main graph that is modified with the saved configuration has the same view style as the metrics in the graph you save the configuration from. Individual style overrides the default style for all metrics.

**Default style**
Default style is the view option that is used to draw metrics in any main graph that is modified with the saved configuration, provided you have not elected to save the individual style. Whether you saved individual style or not, it is also the view style that is used to draw any metrics you add to the main graph after you have applied the configuration. If you do not save the default style, the default style is taken from the X resource **MainPlotStyle**, which defaults to average.

**Current scaling**
If you save the current scaling of the metrics in the main graph, metrics in any main graph that is modified with the saved configuration uses the same scale as used in the main graph you save the configuration from.

**Current colors**
Color assignment to the metrics can only be saved if you do not decide to save the configuration as host-independent (see the following). The reason is that when a host-independent configuration is used to change the view of a main graph that includes metrics from multiple hosts, the same metric name may exist for more than one host. A fixed color assignment would make the graph difficult to read.

**Current graph size**
If you elect to save the size of the main graph, the window of any main graph that is modified with the saved configuration is resized to become the same size as the window of the main graph you save the configuration from.

**Host dependency**
Usually, metric names include the name of the host where the metrics were collected. When you save a configuration you can elect to save it so the host-dependent part is not saved. This allows you to apply the saved configuration to any main graph, no matter which host produced the recording and even when some metrics are displayed for more than one host. If you do not save the configuration as host-independent, it can only be used to view metrics in the exact same host/metric combinations as those of the main graph you save the configuration from.

## Applying Configurations

The Replace Configuration dialog box is displayed when you drag the **Config** icon to a main graph. The dialog box contains a list of all the configurations available in your configuration file. If necessary, a scroll bar allows you to scroll in the list of configurations.

To use a named configuration to customize the main graph, you can use either of the following methods:

1. Select a line from the list and then select **OK**. This customizes the main graph with the selected configuration and closes the dialog box.
2. Drag one of the lines to the main graph and drop it there. The configuration is used to customize the main graph and the dialog box stays open so you can repeat this operation.

If you decide not to change the view of the main graph, select **Cancel**. Selecting **Cancel** after you have dragged a configuration line to the main graph does not undo the effect of the drag,

Each of the lines in the list can also be drawn to the **Pit** icon. Each time you do this, the corresponding configuration is deleted from the configuration file and the list is updated.

Because the Replace Configuration dialog box stays open after you have dragged a configuration line to a main graph, it is possible to drag configuration lines to other main graphs than the one you dragged the **Config** icon to when the dialog box opened. Because this seems like a useful facility, it is permitted to do so. It has the side effect, though, that whenever you select **OK**, the selected configuration line is applied to the last main graph you dragged a configuration line to. Because of this, always select **Cancel** to close the dialog box if you have dragged configuration lines from the box to one or more main graphs.

### Deleting Configurations

Named configurations can be deleted from the Replace Configuration dialog box as explained in the section "Applying Configurations" on page 124. They can also be deleted from a dialog box titled Delete Configuration. This dialog box opens when you drag the **Config** icon to the **Pit** icon. Both the Replace Configuration dialog box and the Delete Configuration dialog box contain a list of all configurations in your configuration file.

To delete a configuration view, drag the corresponding line in the dialog box to the **Pit** icon. The configuration is deleted and the list of configurations in the dialog box is updated.

When you have no more deletions to do, close the dialog box by selecting **Cancel**.

### Techniques for Using Configuration Views

A recording file frequently contains more statistics than it is practical to view in a single main graph. The **azizo** configuration file, when properly customized, allows multiple views of the same data simultaneously.

When applying configuration views, realize that the top-level main graph is unlikely to include all the metrics from the recording file. Because of this, any configuration view you apply to the top-level main graph may fail to show the metrics that were filtered out when the main graph was created. This same is true if you apply a configuration view to a main graph that you applied a different configuration view to earlier.

Therefore, before applying a configuration view, create another main graph by zooming in on whichever time period you are interested in and selecting **Rescan**. After the new main graph is displayed, apply the configuration view to that graph.

For multiple views of the same data simultaneously, create as many zoomed-in main graphs as you want views. Then apply different configuration views to each main graph. For easy correlation, let all of the zoomed-in main graphs cover the same time period.

The supplied sample configuration file in **/usr/samples/perfmgr/azizo.cf** has configurations defined for different views of recordings from the Local System Monitor and Comby Style Sample Consoles and the Multi-host Monitor and Single-host Monitor skeleton consoles defined in the distributed **xmperf** configuration file.

## Common azizo Dialog Boxes

Some of the **azizo** dialog boxes are identical or similar for multiple object types. Such dialog boxes are described in the following sections.

## The Print Box

The Print Box opens when you drag a metrics graph from the metrics selection window, the title part of the metrics selection window, or a main graph to the **Print** icon and drop it there. In other words, whenever you want to print a graph. The Print Box is used to control printing of the graph. It is divided in two by a vertical separator.

The left part of the dialog box has three sections:

**Graph Description**
Allows you to enter a text that is printed below the graph.

**Available Printers**
Allows you to select the printer where you want the summary report to be printed. This field is ignored if you enter a file name in the next input field. The list of available printers is derived from the printers defined on your system.

When print is directed to a printer, **azizo** uses a print command defined by the X resource **PrintCommand**. The command defaults to **lp -d**. The resulting command line used to print on a printer called `psprint` would then be **lp -d psprint** followed by a temporary file name created by **azizo**.

**File name if not direct print**
If you don't want the printed output to go directly to a printer, enter a file name in this field. The print output is written to the specified file name.

The right part of the dialog box is divided in two by a horizontal separator. The top section has a series of on/off options. In the bottom section, you can enter numerical values to control the printing. The on/off options are as follows:

**Reverse Colors**
If selected, inverts all colors when producing the PostScript output. This option is useful if your graph has a dark background, which often gives inferior print quality and reduced legibility.

**Check Printer Memory**
If selected, causes **azizo** to check if the selected printer has sufficient memory to print the image. Because the print image is sent to the printer as a background job, you are not warned if the image is too large, but the print output is orderly terminated with a message that states the reason.

**Use Color PostScript**
If your printer supports color PostScript, select this option. It gives you superior quality.

**Landscape** If selected, prints the graph in landscape format.

**Generate Binary Image**
Reduces the size of the PostScript output file. Recommended if you print to a file and if your printer supports this format. Try it out to make sure.

**Encapsulate** If selected, wraps the PostScript output in EPSF (Encapsulated PostScript Format) tags and suppresses PostScript commands to position the image and scale it. Use this option if you want to imbed the generated PostScript output in another PostScript document. The generated output, as a rule, is not directly printable.

**Compress output**
Uses the standard UNIX compress command to compress the output from the print function. Recommended because it saves disk space. Some printers may not be able to print this format.

The options that accept numeric input in the bottom part are:

**Brighten factor**
Allows you to modify the brightness of the generated print image. The factor is used to brighten or darken the image. 100 is the base and a larger number brightens the image while a smaller number darkens the image. Permitted range of the brighten factor is 0 through 200. For black and white printing, a brightness factor of 100 is recommended. The initial default for brighten factor is set with the X resource **BrightenFactor**, which has a default of 100.

**Paper width**
Specifies the width of the paper you are using. The width is given in inches with two decimal positions. The initial default for paper width is set by the X resource **PaperWidth**, which defaults to 8.5 inches.

**Paper height**
Specifies the height of the paper you are using. The height is given in inches with two decimal positions. The initial default for paper height is set by the X resource **PaperHeight**, which defaults to 11 inches.

**Horizontal margins**
Specifies the horizontal margins to use when scaling the graph. This value determines the minimum distance between the graph and the left and right edges of the paper. The printed graph is always centered on the page. Margins are given in inches with two decimal places. The initial default for horizontal margin is set by the X resource **HorizontalMargin**, which defaults to 0.5 inch.

**Vertical margins**
Specifies the vertical margins to use when scaling the graph. This value determines the minimum distance between the graph and the top and bottom edges of the paper. The printed graph is always centered on the page. Margins are given in inches with two decimal places. The initial default for vertical margin is set by the X resource **VerticalMargin**, which defaults to 0.5 inch.

When you have selected the print options you want, select **OK** to proceed. To cancel the print, select **Cancel**.

### How Graphs Are Printed

Printing of graphs is done by capturing the image of the graph as it opens on your display and converting that image to PostScript format. To ensure that the image is correctly captured, the first thing to do when planning to print a graph is to position the graph window so that it is fully visible and does not obscure the **Print** icon.

To initiate the printing, drag the graph to the Print icon. This causes the Print Box to open. From the dialog box you set the options as explained earlier. After you select the **OK** button in the Print Box, the dialog box goes away and **azizo** waits two seconds before it begins the capture of the graph image you want to print.

The two-second delay allows the X server to refresh the window so that any part of the graph that was obscured by the dialog box is also refreshed. When the two seconds have elapsed, you hear a beep. This beep signals the start of the capture of the graph image. When the capture is completed, another beep sounds.

Anything you do on your screen in the time between the two beeps can potentially cause the print image to be obscured so the output generated by the image capture is corrupted. Therefore, do not use the mouse or keyboard until you hear the second beep.

## The Report Box

The Report Box opens when you drag the data part of an information window to the **Print** icon and drop it there. In other words, whenever you want to print the tabular view of the statistical data for a main graph or the metrics selection window. The Report Box is used to control printing of the report.

The dialog box has three parts:

**Print Heading** Allows you to enter a text string that is printed as the heading on all pages produced.

**Available Printers**
Allows you to select the printer where you want the summary report to be printed. This field is ignored if you enter a file name in the next input field. The list of available printers is derived from the printers defined on your system.

When print is directed to a printer, **azizo** uses a print command defined by the X resource **PrintCommand**. The command defaults to **lp -d**. The resulting command line used to print on a printer is called `ascprint` would then be **lp -d ascprint** followed by a temporary file name created by **azizo**.

**File name if not direct print**
If you don't want the printed output to go directly to a printer, enter a file name in this field. The print output is written to the specified file name.

After you have changed the fields in the dialog box, select **OK** to execute the printing. If you don't want to go through with the printing, select **Cancel**.

The generated print file is a plain ASCII file. When you print the information window of the metrics selection window or the top-level main graph, the print output contains only one section that lists the statistical data for all the metrics in the object. If the object you drag to the **Print** icon is a zoomed-in main graph, the printed report contains two parts: one for the entire time period covered by the recording file; another for the zoomed-in time period. Each part may contain multiple pages.

```
All Metrics     Page 1
Summary for graph 100%, Dec 20 13:32:19 1993-Dec 20 15:49:10 1993
No of Average Standard
Maximum Minimum
Observ  Value  Deviation  Value  Value  Metric pathname
------  -----  ---------  -----  -----  ------------------
  774   0.87    3.61      44.80   0.00  hosts/snook/Disk/hdisk0/wblk
  774   0.00    0.06       1.60   0.00  hosts/snook/Disk/hdisk0/rblk
  774   0.09    0.32       4.20   0.00  hosts/snook/Disk/hdisk0/xfer
  774   0.11    0.40       5.80   0.00  hosts/snook/Disk/hdisk0/busy
  721   1.77    7.37      89.63   0.00  hosts/nchris/Disk/hdisk1/wblk
  721   0.34    4.19     108.87   0.00  hosts/nchris/Disk/hdisk1/rblk
  721   0.22    0.85      11.41   0.00  hosts/nchris/Disk/hdisk1/xfer
  721   0.49    1.93      26.42   0.00  hosts/nchris/Disk/hdisk1/busy
  721  29.96   96.66     875.38   0.00  hosts/nchris/Disk/hdisk0/wblk
  721   6.83   87.48    2248.02   0.00  hosts/nchris/Disk/hdisk0/rblk
  721   1.73    3.97      49.80   0.00  hosts/nchris/Disk/hdisk0/xfer
  721   3.42    7.79      86.80   0.00  hosts/nchris/Disk/hdisk0/busy
  777   2.82   10.10      91.20   0.00  hosts/drperf/Disk/hdisk0/wblk
  777   2.95   42.95    1070.44   0.00  hosts/drperf/Disk/hdisk0/rblk
  777   0.46    1.67      19.80   0.00  hosts/drperf/Disk/hdisk0/xfer
  777   1.04    3.91      46.20   0.00  hosts/drperf/Disk/hdisk0/busy


37.68%      Page 1
Summary for graph 44.66%, Dec 20 13:33:00 1993 - Dec 20 14:34:07
1993

No of   Average    Standard     Maximum   Minimum
Observ  Value      Deviation    Value     Value     Metric path name
------  -------    ---------    -------   -------   ----------------
619      28.44      90.47        875.38    0.00     hosts/nchris/Disk/hdisk0/wblk
619       7.89      94.38       2248.02    0.00     hosts/nchris/Disk/hdisk0/rblk
619       1.76       4.06         49.80    0.00     hosts/nchris/Disk/hdisk0/xfer
619       3.45       7.79         86.80    0.00     hosts/nchris/Disk/hdisk0/busy
         100%      Page 2
Summary for graph 100%, Dec 20 13:32:19 1993 - Dec 20 15:49:10 1993
No of   Average    Standard     Maximum   Minimum
Observ  Value      Deviation    Value     Value     Metric path name
```

```
------  -------   ---------   -------   -------   ----------------
721       29.96     96.66      875.38    0.00     hosts/nchris/Disk/hdisk0/wblk
721        6.83     87.48     2248.02    0.00     hosts/nchris/Disk/hdisk0/rblk
721        1.73      3.97       49.80    0.00     hosts/nchris/Disk/hdisk0/xfer
721        3.42      7.79       86.80    0.00     hosts/nchris/Disk/hdisk0/busy
```

## Changing View Options Dialog Box

The purpose of the Changing View Options dialog box is to allow you to change the way one or all metrics are plotted in an object. The dialog box is displayed when you drag one of the following object types to the **View** icon:

**A main graph** The new style is used for all metrics in the main graph.

**A metric label**
The new style is used only for the metric corresponding to the metric label that was dragged to the **View** icon.

**Title part** When the object is the title part of the metrics selection window, the new style is used for all metrics graphs in the metrics selection window.

**A metrics graph**
The new style is used only for the metrics graph that was dragged to the **View** icon.

The dialog box has two incarnations. The first is displayed when the drag object comes from a main graph; the second when the object is part of the metrics selection window. The dialog box identifies the object you dragged to the **View** icon on the first text line in the box. The next two lines further point out the type of the selected object. The remaining lines give a brief explanation of the available view styles. The row of buttons at the bottom allows you to select the view style you want or to keep the current view style by selecting **Cancel**.

To understand how the view style is affected by the available data, consider that each metric is drawn from an array of observations where each element in the array corresponds to a time interval into which a single observation, multiple observations, or none of those in the recording file may fit. Only if the time interval covers multiple observations can there be a difference between plotting the maximum value, the minimum value, or the average value in the interval.

The incarnation of the dialog box when the object is a main graph or a single metric label from a main graph is has the following view types, all of which are drawn as line graphs:

**Average** The metrics are drawn from the average observation value in each of the intervals.

**Maximum** The metrics are drawn from the highest observation value in each of the intervals.

**Minimum** The metrics are drawn from the lowest observation value in each of the intervals.

**Both** Two lines are drawn, one from the highest observation value in each of the intervals, the other from the lowest observation value in each of the intervals.

If the object is a single metrics graph from the metrics selection window or the title part of the metrics selection window, the view types available are:

**Line, Maximum**
The metrics are drawn as a line graph, using the highest observation value in each of the intervals.

**Line, Max & Min**
The metrics are drawn as a line graph. For each metric, two lines are drawn, one using the highest observation value in each of the intervals, the other using the lowest observation value in each interval.

**Bar, Max-to-Min**
The metrics are drawn as vertical lines (bars) in each of the intervals that have

observations. The vertical line is drawn from the lowest observation value to the highest observation value in the interval. This gives a graphical representation of the variance of the observations. If the time interval covered by each of the interval slots in the metrics graphs is less than one second, attempts to select this drawing style reverts to the Line, Max & Min style.

## Overview of Valid Drag-and-Drop Operations

The drag and drop interface of **azizo** may be new and strange to some users. This section is intended to be used as a reference section and may be useful to such users. It contains subsections for each of the objects and actions defined by **azizo**. Where applicable, each subsection describes:

1. What can be dragged to the object or action.
2. Where the object or action can be dropped.

## Annotation Icon

The **Annotate** icon, available with Version 2.2 or later, is used to add or modify annotation text to a recording. Annotation text is kept in a separate file, linked to the recording file by a naming convention, as explained in “Annotation Files” on page 95.

### What Can Be Dragged to the Annotate Icon

Only main graphs can be dragged to the **Annotate** icon. When you drop the main graph, an editor window will open. From the editor window, you can modify any existing annotation text or create a new annotation file, if none existed before. When you have made the changes you want, exit the editor window with the **file** or **save** option of the editor in use. If you exit the editor with the **quit without saving** option of the editor in use, any existing file is left unchanged and no new annotation file is created if none existed before.

### Where Can You Drop the Annotate Icon

The **Help** icon displays a help text.

## Config Icon

The **Config** icon is used to save, retrieve, and delete customized views of main graphs. The idea is that by saving a particular view, you can later use it for viewing other recordings and display the graphs in a way that allows you to immediately compare the data in the recordings. Each saved view is called a configuration. The **Config** icon is used to maintain and use configurations.

### What Can Be Dragged to the Config Icon

Only main graphs can be dragged to the **Config** icon. When you drop the main graph, the dialog box Writing Configuration opens. You can then set the properties you wish to save and give a name to the configuration.

### Where Can You Drop the Config Icon

**Main graphs**
The Replace Configuration dialog box is displayed. From that dialog box you can select a configuration to customize the view of the data currently displayed in the main graph.

**Pit Icon**
The Delete Configuration dialog box is displayed. From that dialog box you can drag individual configurations to the **Pit** icon to delete them.

**Help Icon**
Displays a help text.

## Configuration Lines

Configuration lines are the detail lines in the dialog boxes Delete Configuration and Replace Configuration. Each line represents a named configuration view as it is saved in the configuration file. To use a configuration line as a drag object, place the pointer on the line and start the drag operation from there. Nothing can be dropped on configuration lines.

### Where Can You Drop a Configuration Line

**Main Graph**
When the configuration line resides in the Replace Configuration dialog box, you can drag it to and drop it on a main graph. Each time, the configuration dropped is used to customize the view of the main graph.

**Pit Icon**
When either of the Delete Configuration or Replace Configuration dialog boxes is displayed, individual lines can be dragged to the **Pit** icon to delete them from the configuration file.

**Help Icon**
Displays a help text.

## Exit Icon

The Exit icon allows you to exit the **azizo** program. The only drag-and-drop operation permitted for this icon is to drag the icon to the **Help** icon to display a help text.

## Filter Icon

The **Filter** icon is used when you want to produce a filtered recording file as a subset of the current recording file. The only place where the **Filter** icon can be dropped is the **Help** icon.

### What Can Be Dragged to the Filter Icon

If you drop a main graph on the **Filter** icon, the dialog box titled Writing Filtered Recording opens. Filtering is the process of creating a copy of the recording file you are currently analyzing, using the current main graph to select the metrics you want the copy of the recording file to contain. The dialog box allows you to give the filtered recording file a name and to change the time interval it covers from the default displayed in the dialog box. The default is the time interval covered by the main graph you dragged to the **Filter** icon.

## Help Icon

The **Help** icon is the entry point to the help facility. It is the only icon that cannot be dragged anywhere. If you select the **Help** icon, the help index is displayed.

### What Can Be Dragged to the Help Icon

All objects and all other icons can be dragged to the **Help** icon to display the help text for the object or icon.

## Info Icon

The **Info** icon is used to display summary information for either a single metric or all metrics in a main graph or in the metrics selection window. The only place where the **Info** icon can be dropped is the **Help** icon. Doing so gives you a help text for the **Info** icon.

### What Can Be Dragged to the Info Icon

**Main graphs**
When you drop a main graph on the **Info** icon, an information window opens. If the main graph is the top-level main graph, the information window includes summary information for the entire time period covered by the recording. If the main graph is a zoomed-in graph, one set of summary information is shown for the zoomed-in time period and one for the full time interval covered by the recording. In either case, one line is shown for each of the metrics in the main graph.

**Metric label from a main graph**
When you drag one of the metric labels of a main graph and drop it on the **Info** icon, an information window opens. If the main graph is the top-level main graph, the information window includes summary information for the entire time period covered by the recording. If the main graph is a zoomed-in graph, the summary information is shown for the zoomed-in time period. Only information about the selected metric is shown.

**Title part of the metrics selection window**
: When you drag the title part of the metrics selection window and drop it on the **Info** icon, an information window opens. The information window includes summary information for the entire time period covered by the recording. The window contains one line for each of the metrics in the metrics selection window.

**Metrics graph from the metrics selection window**
: When you drag one of the metrics from the metrics selection window and drop it on the **Info** icon, an information window opens. The information window includes summary information for the entire time period covered by the recording for the selected metric.

# Information Window

Information windows that are generated from a main graph or from all the metrics in the metrics selection window can be used as objects. No objects can be dropped on information windows.

## Where Can You Drop an Information Window

**Print Icon**
: Causes the Report Box dialog box to open. From this dialog box you can print a report with the data as shown in the information window.

**Help Icon**
: Displays a help text.

# Local Files Icon

This icon is used to select a recording file to analyze. When you select the icon, the Select Recording File selection box is displayed. No objects can be dragged to the Local Files icon. The only place where the Local Files icon can be dropped is the **Help** icon. Doing so gives you a help text for the Local Files icon.

# Main Graphs

Main graphs can be used as objects that can be dragged to and dropped on actions. Selected other objects can be dragged to main graphs. To drag a main graph, start the drag operation with the pointer in the graph section of the main graph window. Similarly, when objects are dropped onto a main graph, the point where you drop the object must be within the graph area.

## What Can Be Dragged to a Main Graph

**Metrics graph**
: Drag a metrics graph from the metrics selection window to the main graph to add that metric to the graph. If the metric already is part of the main graph, this is handled as a no-operation. When the metric is added to the main graph, it is given the next free color.

**Config Icon**
: When you drop the **Config** icon on a main graph, the configuration file is searched for available configurations. If such exist, a dialog box opens and gives a list of all configuration names. You can then select which configuration view to apply to the main graph. When the configuration name is selected, select **OK** to apply it to the graph; select **Cancel** to leave the graph unchanged. Alternatively, you can drag a configuration line to the main graph as explained below.

**Configuration Line**
: When the Replace Configuration dialog box is displayed, you can drag individual lines from the list of configurations and drop them on the main graph. Each time, the configuration dropped is used to customize the view of the main graph. The dialog box stays up until you close it by selecting **Cancel**.

## Where Can You Drop a Main Graph

**Annotate Icon**
: Beginning with Version 2.2, when you drop a main graph on the Annotate icon, an editor window will open. From the editor window, you can modify an existing annotation text or

create a new annotation file if none existed before. When you have made the changes you want, exit the editor window with the **file** or **save** option of the editor in use. If you exit the editor with the **quit without saving** option of the editor in use, any existing annotation file is left unchanged and no new annotation file is created if none existed before.

**Config Icon**
When you drop a main graph on the **Config** icon, the Writing Configuration dialog box opens. You can then set the properties you wish to save and give a name to the configuration. Select **OK** to save the configuration view to the configuration file.

**Info Icon**
When you drop a main graph on the **Info** icon, an information window opens. If the main graph is the top-level main graph, the information window includes summary information for the entire time period covered by the recording. If the main graph is a zoomed-in graph, one set of summary information is shown for the zoomed-in time period and one for the full time interval covered by the recording. In either case, one line is shown for each of the metrics in the main graph.

**Print Icon**
Dropping a main graph on the **Print** icon causes the Print Box dialog box to pop open. From this box you select the options to use when printing the graph and enter the description you want printed for the graph. You can select to print directly to one of the printers known by your system, or you can send the print image to a file. Print images are in PostScript format.

**Filter Icon**
If you drop a main graph on the **Filter** icon, the Writing Filtered Recording dialog box opens. Filtering is the process of creating a copy of the recording file you are currently analyzing, using the current main graph to select the metrics you want the copy of the recording file to contain. The dialog box allows you to give the filtered recording file a name, and to change the time interval it covers from the default displayed in the dialog box. The default is the time interval covered by the main graph you dragged to the **Filter** icon.

**View Icon**
Dragging a main graph to the **View** icon displays a dialog box titled Changing View Options. The purpose of this dialog box is to allow you to change the way metrics are plotted in the main graph. The view style you select becomes the default view style for the main graph.

**Scale Icon**
When you drop a main graph on the **Scale** icon, you see a dialog box titled Rescaling from which you can change the scale of all the metrics in the main graph. You can elect to use **Autoscale**, which adjusts the scales of all metrics so the maximum value in the time interval covered by the main graph falls within the 50% and 100% lines in the graph. If you select **Normscale** all metrics revert to the scale contained in the recording file.

**Pit Icon**
By dropping a main graph on the **Pit** icon, you cause the main graph and all its descendants (all main graphs created by zooming-in on the dragged graph) to be deleted. If the dragged graph is the top-level main graph, this graph itself is not deleted but all metrics are removed from it.

**Help Icon**
Displays a help text.

## Metric Label from Main Graph

A metric label represents a single metric of a main graph. Any action performed when dropping a metric label onto an icon affects only the metric represented by the metric label and only in the main graph from where the metric label is dragged. Nothing can be dropped on a metric label.

### Where Can You Drop a Metric Label

**Info Icon**
When you drop one of the metric labels of a main graph onto the Info icon, an information window opens. If the main graph is the top-level main graph, the information window includes summary information for the entire time period covered by the recording. If the main graph is a zoomed-in graph, the summary information is shown for the zoomed-in time period. Only information about the selected metric is shown.

**View Icon** Dragging a metric label of a main graph to the **View** icon displays a dialog box titled Changing View Options. The purpose of this dialog box is to allow you to change the way the dragged metric is plotted in the main graph.

**Scale Icon** When you drop a metric label of a main graph on the **Scale** icon, you see the Rescaling dialog box from which you can change the scale of the dragged metrics in the main graph. You can elect to use **Autoscale**, which adjusts the scale of the metric so the maximum value in the time interval covered by the main graph falls within the 50% and 100% lines in the graph. If you select **Normscale** the metric reverts to the scale contained in the recording file.

**Pit Icon** By dropping a metric label from a main graph on the **Pit** icon, you delete the metric from the main graph.

**Help Icon** Displays a help text.

## Metrics Graph

A metrics graph is the graphical representation of the values for a single statistic as shown in the metrics selection window. The metrics graph is intended to give users a visual representation of how the observed values for the statistic have varied over the time period covered by the graph. When a metrics graph is dropped on an action, the action is applied only to the metric represented by the metrics graph. Other metrics in the metrics selection window are unchanged. No objects can be dropped on a metrics graph.

### Where Can You Drop a Metrics Graph

**Main Graph** When you drag a metrics graph icon from the metrics selection window to a main graph, you tell the **azizo** program that you want to add the corresponding metric to the main graph. If the metric is not already included in the main graph, it is added to the graph and the main graph is redrawn. When the metric is added to the main graph, it is given the next free color. If the metric already was part of the main graph, this action is handled as a no-operation.

**Info Icon** When you drag a metrics graph to the **Info** icon and drop it there, an information window opens. The information window includes summary information for the entire time period covered by the recording for the selected metric.

**Print Icon** Dropping a metrics graph on the **Print** icon tells **azizo** that you want to print the graphical image of the metrics graph and causes the Print Box dialog box to pop up. From this box you select the options to use when printing the graph and enter the description you want printed for the graph. You can select to print directly to one of the printers known by your system or you can send the print image to a file. Print images are in PostScript format.

**View Icon** Dragging a metrics graph from the metrics selection window to the View icon displays a dialog box called Changing View Options. The purpose of this dialog box is to allow you to change the way the metric is plotted in the metrics selection window.

**Pit Icon** Dragging a metrics graph to the **Pit** icon causes the graph to be removed from the metrics selection window. Consequently, you can no longer use the metric for any purpose, such as adding it to a main graph. However, the recording file is not modified and you can always start over by rereading the recording file. The metric is also deleted from any main graph it was part of.

Even though the metric no longer seems to be available, it is not completely forgotten: If you zoom-in on a main graph and select the **Rescan** option, then the recording file is scanned for all metrics that have observations within the zoom-in outline. If the metric you deleted from the metrics selection window meets the criterias for being included in the zoomed-in graph, then it is included regardless of it no longer being accessible from the metrics selection window.

**Help Icon** Displays a help text.

## Metrics Selection Window

The metrics selection window consists of a title part followed by a list of metrics graphs. The title part is an object and can be dropped on various actions. When this is done, the action is performed on *all the metrics* in the metrics selection window. No objects can be dropped on the metrics selection window.

### Where Can You Drop the Title Part

**Info Icon**
When you drag the title part of the metrics selection window to the **Info** icon and drop it there, an information window opens. The information window includes summary information for the entire time period covered by the recording for all metrics in the metrics selection window.

**Print Icon**
Dropping the title part of the metrics selection window on the Print icon is interpreted as a request to print the graphical image of the metrics graphs currently visible in the metrics selection window. It causes the Print Box dialog to opens. From this box, select the options to use when printing the graph and enter the description you want for the graph. You can select to print directly to one of the printers known by your system, or you can send the print image to a file. Print images are in PostScript format.

The generated print image contains the entire metrics selection window, including scroll bars, if present, and including the title part of the window. Only those metrics graphs visible in the window when printing starts are included in the print image.

**View Icon**
Dragging the title part of the metrics selection window to the View icon displays the Changing View Options dialog box whose purpose is to allow you to change the way metrics are plotted in the metrics selection window. Any selection of style you make affects all metrics in the metrics selection window.

**Help Icon**
Displays a help text.

## Pit icon

The **Pit** icon is where you drop objects you no longer need. This causes the object you drop to be removed from where you dragged it but never causes changes to the recording file. Thus, removal of objects is only from the viewing environment you are in. It is always possible to start over by re-reading the recording file. When you drag configuration lines to the **Pit** icon, however, you permanently delete the named configuration from the configuration file. The only place where the **Pit** icon can be dropped is the **Help** icon.

### What Can Be Dragged to the Pit Icon

**Config Icon**
The Delete Configuration dialog box is displayed. From the dialog box you can drag individual configurations to the **Pit** icon to delete them from the configuration file.

**Configuration lines**
When either of the Delete Configuration or Replace Configuration dialog boxes is displayed, individual lines can be dragged to the **Pit** icon to delete them from the configuration file.

**Main graph**
By dropping a main graph onto the **Pit** icon, you cause the main graph and all its descendants (all main graphs created by zooming-in on the selected graph) to be deleted. If the selected graph is the top-level main graph, this graph itself is not deleted but all metrics are removed from it.

**Metric label from main graph**
By dropping a metric label of a main graph on the **Pit** icon, you delete the metric from the main graph.

**Metrics graph from metrics selection window**
If you drag one of the metrics graphs from the metrics selection window to the **Pit** icon

and drop it there, the corresponding metric is deleted from the metrics selection window. In addition, the metric is deleted from all main graphs.

The metric cannot be added back to the metrics selection window except by rereading the recording file. However, if you use the **Rescan** option when zooming in on a main graph, the deleted metric again is included in the scan, because the scan is done on the actual file rather than from the metrics selection window.

# Print Icon

The **Print** icon is where you drop objects you want to print. This causes either the Print Box or the Report Box to open. The former when the object is a graph; the latter when the object is an information window. The only place where the **Print** icon can be dropped is the **Help** icon.

When printing graphs, **azizo** first captures the image of the graph, then converts the image to PostScript format, and finally sends the print file to a printer or a file. The capture of the image depends on the graph being visible on the screen as explained in the section “How Graphs Are Printed” on page 127.

## What Can Be Dragged to the Print Icon

**Main Graph**
Used to print the image of an entire main graph window. Causes the Print Box dialog box to open.

**Title part of metrics selection window**
Used to print the image of the metrics selection window exactly as it is displayed in the main window. The entire window including scroll bars, metrics graphs, and title part is printed. The Print Box allows you to customize the printing.

**Metrics graph from the metrics selection window**
Used to print the image of a single metrics graph from the metrics selection window. The Print Box allows you to customize the way the graph is printed.

**Information window**
Used to print the contents of an information window. Only information windows for main graphs and the entire metrics selection window can be printed. The Report Box allows you to customize the way the data is printed.

# Scale Icon

The **Scale** icon is used to change the scale for one or more metrics in a main graph. This is done by dragging an object to the icon. The only place where the **Scale** icon can be dropped is the **Help** icon.

## What Can Be Dragged to the Scale Icon

**Main Graph**
When you drop a main graph onto the **Scale** icon, you see the Rescaling dialog box from which you can change the scale of all the metrics in the main graph. You can elect to use **Autoscale**, which adjusts the scales of all metrics so the maximum value in the time interval covered by the main graph all fall within the 50% and 100% lines in the graph. If you select **Normscale**, all metrics revert to the scale contained in the recording file.

**Metric label from Main Graph**
When you drop a metric label of a main graph onto the **Scale** icon, you get the Rescaling dialog box from which you can change the scale of the selected metric in the main graph. You can elect to use **Autoscale**, which adjusts the scale of the metric so the maximum value in the time interval covered by the main graph falls within the 50% and 100% lines in the graph. If you select **Normscale** the metric reverts to the scale contained in the recording file.

## View Icon

The **View** icon is used to change the way one or more metrics is plotted in the graph where it is displayed. This is done by dragging a metrics object, a main graph, or the title part of the metrics selection window onto the icon. The only place where the **View** icon can be dropped is the **Help** icon.

### What Can Be Dragged to the View Icon

**Main Graph**
Dragging a main graph to the **View** icon displays the Changing View Options dialog box. The purpose of this dialog box is to allow you to change the way metrics are plotted in the main graph. The view style you select becomes the default view style for the main graph.

**Metric label from Main Graph**
Dragging a metric label of a main graph to the **View** icon displays the Changing View Options dialog box, which allows you to change the way the selected metric is plotted in the main graph.

**Title part of the metrics selection window**
When you drop the title part of the metrics selection window onto the **View** icon, the Changing View Options dialog box opens. From this dialog box you can change the plotting method for all the metrics in the window in one operation.

**Metrics graph from the metrics selection window**
Dragging a metrics graph from the metrics selection window to the View icon displays the Changing View Options dialog box. The dialog box allows you to change the way the metric is plotted in the metrics selection window.

# Chapter 11. Analyzing Performance Trend Recordings with the jazizo Tool

This section discusses the following jazizo topics:


- “Recording Files”
- “Configuration Files” on page 140
- “Jazizo Tool Menus” on page 140
- “Legend Panel” on page 145


**Jazizo** is a tool for analyzing the long-term performance characteristics of a system. It analyzes recordings created by the **xmtrend** daemon, and provides displays of the recorded data that can be customized. The **jazizo** tool can be configured to show only the data of interest, in concise graphical or tabular formats. Users can create, edit, and save custom configurations. In addition, reports can be generated covering specific time periods, and data-reduction options are provided to assist analysis.

## Recording Files

A recording file contains trend metric values recorded by **xmtrend**. The trend metrics are timestamped and more than one file may be used to record data over time. If **xmtrend** was directed to record for three months and create one file per month, then the **jazizo** tool would use three recording files to graph trends across the three months. From the command line, either a specific recording file could be specified or a directory where the recording files reside. Recording files can also be selected from the **jazizo** menus.

## Trend Statistics (Metric) Definitions

Trend statistics are called *trend metrics*. These are system values such as free memory or *%CPU Idle* that **xmtrend** gathers from the system and converts to statistical values such as mean, maximum, minimum, and standard deviation. In other words, **xmtrend** may monitor a metric every second and compile these observations into a set of statistics once per hour. In this fashion, large amounts of data can be recorded over long periods of time. See “Starting Recording Sessions from the xmtrend Command Line” on page 175 for further explanation of the **xmtrend** tool.

Trend metrics gathered by **xmtrend** have various attributes to help interpret the data. Some of these attributes can be modified by **jazizo** to further refine analysis. The attributes are **hiRange** and **loRange**.

### hiRange and loRange

These values are interpreted by the graph as being the lowest and highest values to show. Although the data may be outside of this range, the graph will not display those values. Usually, the **loRange** and **hiRange** contain all the data points. These values can be changed per metric from the Metric Selection menu from the Legend Panel. The new settings can be saved to a configuration file.

For example, a metric such as *%CPU Idle* would have a **loRange** of 0, and a **hiRange** of 100, because it represents a percent.

### Recording Frequency

Recording frequency is how often a trend value is recorded. It should not be confused with sampling frequency, which represents the rate at which values from the system are sampled by the performance agents. Although the minimum recording frequency is once per minute, **xmtrend** samples the data at much higher frequencies (typically once per second). Trend metrics are statistical measurement (mean, maximum, minimum and standard deviation) gathered over the recording period. Internal sampling frequencies are determined by the agent, and cannot be configured by the user. Although **xmtrend** can record trend values at one frequency, **jazizo** can further reduce the data to a different display frequency, so that the number of data points represented on a graph are manageable.

© Copyright IBM Corp. 1994, 2004

### Counter versus Quantity

Metrics are one of two types, *counter* or *quantity*. A counter type measures a value that counts integrally, such as the number of page faults. The value recorded is the delta since the last time it was sampled. A quantity type measures a value that can increase or decrease, such as *%CPU Idle*.

## Configuration Files

Configuration files can be used and customized to select which metrics are graphed, and how they should be displayed. Configuration files can be created and modified by using the menu options in **jazizo**. Because they are text files, they can also be modified manually. A sample configuration file, provided for initialization, is located at **/usr/lpp/perfmgr/jazizo.cf**. To see graphs as defined in the sample configuration file, a trend recording file containing the metrics referenced in the configuration file would need to be available. One configuration file may contain several graph configurations, such as a graph configuration for CPU utilization and another for Memory usage. **Syntax jazizo** configuration files consist of stanzas. Stanzas contain title and optional attribute definitions. Attributes consist of a keyword followed by values. Comments can appear anywhere in the file. Thus, the comments start with `#` and continue to the end of the line. Blank lines are permitted.

Display options can be defined as a default to be applied to all metrics, or to individual metrics. Individual metric stanzas allow specification of which metrics to include, as well as optional attributes for those metrics. In the case where an attribute is specified in both the display options and in individual metric options, the value of the attribute for the individual metric overrides the display options.

The attribute values are specified as strings.

## Jazizo Tool Menus

This section describes the various menus associated with the **jazizo** tool.

There are five types of **jazizo** menus:

1. File menu
2. Edit menu
3. View menu
4. Configuration menu
5. Report menu.

### File Menu

The following options are contained in the File menu:

**Open Recording File**

This option is used to select a recording file if one was not selected from the command line, or a new recording is to be opened. **Jazizo** will display all files that start with `xmtrend` because this is how **xmtrend** names recording files by default. Change the file type to the following to display all files:

```
All Files (*.*)
```

Use **Look in:**, or navigation icons, or by double-clicking on a directory folder to select different recordings. When the recording file is chosen, select **OK** to continue or **Cancel** to exit the menu without selection. When a recording file is selected, the Metric Selection menu is displayed. Metrics and the time frame to be graphed can then be input. See the Edit/Metric-Selection for a view of the Metric Selection menu.

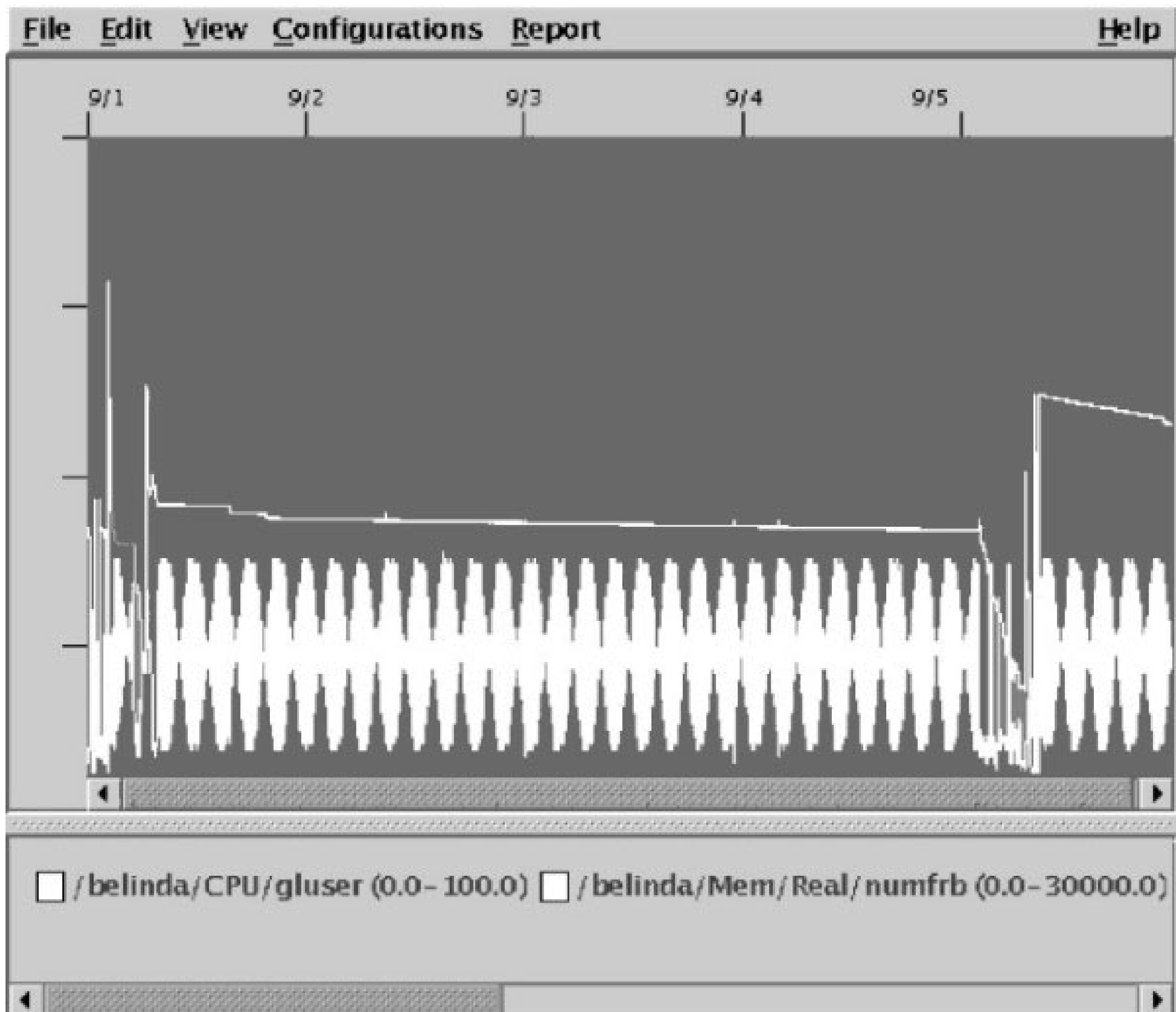



*Figure 6. Jazizo Main Graph Window. This figure shows menu options displayed at the top of the window. The center section is a scrollable graph window showing the activity of two metrics. The top horizontal axis represents time, and the left vertical axis displays a range for the measurements. At the bottom is the metric legend table. Each metric in the table has an associated color box and default range which associate it with the graph. In this case, each metric has a different range, so the vertical axis is only labelled if the metric is selected.*

**Open Configuration**
Configuration files can contain one or more graph configurations. This menu option opens a configuration file and installs the graph configurations into **jazizo**. Navigation and selection of a configuration files works similarly to opening a recording file except that the file suffix is `.cf` for configurations. Unlike recording files where the file can have any name, **jazizo** expects configuration files to end with `.cf` and does not accept any file without this suffix.

**Save Configuration**
Saves current configurations to a configuration file. This provides an alternative to modifying the configuration file by hand.

**Print** Starts a print dialog to print the current graph to either a printer or a Postscript file.

**Exit** Closes all windows and exits **jazizo**.

## Edit Menu

The following options are contained in the Edit menu:

**Metric Selection**

Use this menu option to add or remove metrics that are to be displayed on the graph. The Metric Select window opens automatically when a recording file is loaded from the File menu or entered from the command line with no configuration file.

The metrics on the left of the recording file are those in the Recording, the metrics on the right are to be displayed. Both the available metrics and the selected metrics can be viewed in either tree or list format. Use the tabs to select either the **Tree View** or the **List View**.

Selecting the **Tree View** shows metrics organized into folders and subfolders. These folders are determined by the metrics' classification. A general class of CPU metrics, subclassified into CPU IDs, are further subclassified into type of CPU metric, such as kernel or user. To expand a folder and see its contents, either double-click on its name or single-click on the magnifying-glass icon.

Selecting the **List View** puts every metric on its own line, with no folders for classification. The metrics are listed in alphabetic order.

Highlight the metrics or folders using one of the following methods:

- Single click (one selection)
- Shift and click (more than one selection)
- Ctrl and click (a range of selections)

The metrics can either be added or removed by pressing the **Add** or the **Remove** button. By selecting **Edit** → **Start/Stop** in the Time (X-Axis) section, the start date and time and a stop date and time for all metrics can be selected. **Jazizo** will graph the metric values between the two dates/times. Use this when the recording files are for long periods of time such as a year and only one month's data is needed.

**Graph Properties**

Graph Properties defines how trend metric data is graphed and which statistical values should be graphed. Changes to this menu will be applied to all metrics graphed. Each section of the menu is described in the following section.

**Title** The title of the graph that is synonymous with the configuration name.

**Y-Axis**

**Separate Scales**
Refers to graphing all metrics based on their own scale but having the y-axis scale set to the metric that is selected or hightlighted at the time.

**Common Scale**
**Jazizo** finds a scale that fits all the metrics plotted, generally the highest high and lowest low value among all the metrics selected. All metric data is then graphed to that scale.

**Mean Options**
For trend values, the mean is displayed by default. Standard deviation can also be displayed at the same time.

**Trend Line**
A straight line that represents the overall trend of the mean data. This line is calculated using the Least Squares Best-fit formula.

**All Data**
A trend line is a mathematical best fit line that is drawn on the graph. The Line refers to using all the metric points from the recording to calculate the trend line.

**Visible Data**
This is similar to the Line in that it is a calculated best fit line but only the points within the window will be used for the calculation.

**Off** Turns off the trend line.

**Ranges**
Activates the maximum or minimum values.

**Max Options**
Displays the maximum value for each trend mean sample.

**Min Options**
Displays the minimum value for each trend mean sample.

**OK** Selecting **OK** will apply any changes to all metrics. To set these values individually, refer to the following Metric Properties menu.

**Select A Metric**
A list of selected metrics is displayed from the menu. By selecting a metric, the Metric Properties menu will be displayed. You can change the metric's label, color, and various settings. In addition, the same properties that were described in the "Edit Menu" on page 142 section can be changed, but these changes will only apply to the selected metric. Metric Properties are described further in the "Metric Properties" on page 145 section.

## View Menu

The View menu provides a set of selections for controlling the displayed graph. Selections are based on logical time periods such as All, Year, Quarter, Month, Week, and Day. These selections provide a basic means of filtering the data for viewing. **Jazizo** allows the specification of how much time is displayed in the graph window, and adds scroll bars to view what is not shown.

You can specify the amount of data that will be on the visible graph using All, Year, Quarter, or Month from the View menu. This results in a cascading menu that allows the selection of a corresponding value for the horizontal axis tick increment. For example, if **View** → **Quarter** → **by Month** is chosen, then the visible portion of the graph will represent one quarter-year of data, and each tick on the horizontal axis will represent one month.

For finer granularity, select **Week by Day** or **Day by Hour** from the View menu . In those cases as well, the first unit is the visible X range and the second unit is the tick increment. The visible X range and tick increment affect the behavior of the horizontal scroll bar. Selecting the arrows to the left or right of the horizontal scroll bar will scroll the graph by the value of the tick increment. Selecting inside the scroll bar to the left or right of the slider, the graph scrolls by the value of the visible X range. The scroll bar slider can also be dragged.

The **View** → **Reduce Data by Tick** check box reduces the number of samples in the graph for better manageability. Graph values will be sampled so that there is one data point at every tick mark, but none between the tick marks.

For example, a week might be displayed with day ticks. Looking at the graph lines, one could see that in addition to a point being at every tick, there are also points between the ticks, by selecting **Reduce Data by Tick**, the additional measurements per day would be averaged to 1. In another example, an entire year may be displayed with each tick representing one month. If there are 30 points (days) for every month, and the user wants a monthly average, by selecting **Reduce Data by Tick**, the points will be averaged. Thus, one point will be plotted per month. By deselecting it, the raw data will be graphed.

## Configurations Menu

A **jazizo** graph consists of many options: the metrics that are graphed, their particular options, along with graph-wide options (for example, title of the graph). **Jazizo** allows the storage of multiple configurations

within the Configurations menu. These configurations can be saved or loaded to or from a file using **File Menu Save Configuration**. A configuration file can store multiple graph configurations. Use a predefined configuration by selecting it from the Configuration menu. Configuration names are based on graph titles.

**Add Current Configuration to Menu**
Saves the current configuration as a new configuration. This may include graph properties, metric properties, and which metrics to display. The Title chosen from the Graph Properties menu identifies the configuration in the list.

**Remove A Configuration from Menu**
Removes configurations from the menu. To remove, select a configuration and select **Remove**. To quit without removing a configuration, select **Cancel**.

**Update the Current Configuration**
Saves the current graph properties and metric selection to the currently selected configuration in the menu.

**Select a Configuration**
To select the active configuration (that is, one which applies to the data in the graph), select the name of the configuration from the Configurations menu. Modified, new, and removed configurations are not automatically saved to a file, configurations must be saved before exiting by using **File** → **Save Configuration**.

## Report Menu

### Summary Reports

Displays and prints tabulated reports about the metrics being graphed. The table is organized by individual metrics, and then within each metric into rows that represent the number of samples, the average mean, and the minimum and maximum values. Some Report menu options are:

**Graph** The report generated will be based on all points from the recording file.

**Window**
The report generated will be based on points currently visible in the window.

The report itself is in the form of a table. Each row in the table represents a single metric. There are five columns in the table:

**Name** Contains the metric's label

**Observations**
Shows the total number of data points for that metric

**Average**
Shows the average taken of all the mean values for that metric

**Maximum**
Shows the maximum of all the maximum values for that metric

**Minimum**
Shows the minimum of all the minimum values for that metric

### Selected Metric Reports

Provides a summary of a single highlighted metric. Values for each measurement are displayed with a timestamp.

To print a Report, select **Print**. To close a report, select **Close**.

## Legend Panel

The Legend Panel of the **jazizo** tool is displayed below the graph. It contains a list of all the metrics that are currently selected for the graph. To the left of each metric name is a rectangle showing the metric's color. To the right of each metric is that metric's Y-range on the graph.

The Legend Panel is interactive. Metrics can be manipulated in the graph by left- or right-clicking on the metric in the Legend Panel.

By left-clicking on the metric in the Legend Panel, the metric will be highlighted in the graph. Use this when there are many metrics graphed simultaneously and it is difficult to tell them apart.

Highlighting a metric in the graph changes its color to a bright white. It also changes the Y-Axis labels to reflect the Y-range for this metric. Left-clicking the metric name again deselects it.

Right-clicking on the metric in the Legend Panel starts the legend menu from which the following can be performed:

**Edit Metric**
: Opens a Metric Properties menu. This is the same menu that displays when **Edit** from the main menu bar is selected and a metric label is selected.

**Edit Graph**
: Opens a Graph Properties menu. This is the same menu that displays when **Edit** → **Graph Properties** from the main menu bar is selected.

**Hide or Show the metric in the graph**
: Allows finer control over which metrics are displayed than the Metric Select window. When evaluating a specific set of metrics in a graph, it is sometimes helpful to temporarily hide certain metrics to make others stand out. This can be done by using the Hide and Show choices on this menu.

**Hide-All**
: Hides all metric graphs defined in the legend from the graph. Metrics can then be layered in.

**Show-All**
: Shows all metric graphs defined in the legend from the graph. Metrics can then be layered in.

The **jazizo** tool is keyboard accessible. After a **jazizo** graph is displayed, press Tab to select the first metric in the Legend Panel. Use arrow keys to navigate to other metrics. When a metric is chosen, press Enter to highlight its graph. Press the `M` key to display the legend menu.

Just below the Legend Panel is the status panel. This panel provides general information such as the name of the recording file being used and information messages.

## Metric Properties

Every metric has a group of options specifying how it is displayed within the graph. From the metric properties panel, graphing options can be changed or the graph-wide defaults can be used. The changes apply only to the selected metric. All metrics by default use the overall system setting, configured in the Edit/Graph Properties menu.

Each metric has one Metric Properties window. The panel lists the technical name of the Metric and the user-definable label (the technical name is the default). The user-definable label is displayed in the legend next to the color block and in the metric properties select panel listed previously.

From this panel the color can be changed from a color list. If no **configFile** is loaded that specifies the metric's color, **jazizo** automatically assigns one.

Some metrics are on a percentage scale while others are on the scale of thousands per second, so graphing them together is problematic because the Y-values of the metrics may not be on the same order of magnitude. Although the metrics will be drawn in the same graph, they will all have their own scales (the scales will be displayed in the legend below the graph, next to the name).

Y-range allows the selection of the low and high Y-values on the visible graph, effectively stretching or compressing the data in the vertical direction. This function can also be used to shift the data up or down.

Finally, the options panel is displayed. These values only apply to the metric. The metric can either select its own, or use the user-defined defaults by selecting **Use Default**.

When you are satisfied with the choices, select **OK**. To return to the default values, select **Use Default**. To quit without making changes, select **Cancel**.

# Chapter 12. Analyzing WLM with wlmperf

The following **wlmperf** command is used when analyzing WLM.

## The wlmperf Command

### Purpose

The **wlmmon** and **wlmperf** tools provide graphical views of Workload Manager (WLM) resource activities by class.

### Syntax

**wilmon**

**wlmperf**

### Description

The **wlmmon** and **wlmperf** tools graphically display the resource usage of system WLM activity. The **wlmmon** tool is a disabled version of the **wlmperf** tool, which is part of the Performance Toolbox (PTX) product . The primary difference between the two tools is the period of WLM activity that can be analyzed. The **wlmperf** product can generate reports from trend recordings made by the PTX daemons covering minutes, hours, days, weeks, or monthly periods. The **wlmmon** tool is limited to generating reports within the last 24–hour period.

No usage options exist for the **wlmmon** tool. Three types of visual reports can be generated:

- Snapshot Display
- Detailed Display
- Tabulation Display

The type of report can be customized to cover specified WLM classes over specific time periods. In addition, the WLM activity from two different time periods can be compared (trended) for any chosen display type.

The reports are generated from data that is collected using the same mechanism as the **wlmstat** command. However, this tool uses recordings made by a daemon that must operate at all times to collect WLM data. In the case of **wlmmon**, this daemon is called **xmwlm**, and ships with the base AIX. For the PTX, the **xmtrend** daemon is used to collect and record WLM data.

## Analysis Overview

While the **wlmstat** command provides a per-second view of WLM activity, it is not suited for the long-term analysis. To supplement the **wlmstat** command, the **wlmmon** and **wlmperf** tools provide reports of WLM activity over much longer time periods, with minimal system impact.

The reports generated by this tool are based off samplings made by the associated recording daemon. These daemons sample the WLM and system statistics at a very high rate (measured in seconds), but only record supersampled values at low rate (measured in minutes). These values represent the minimum, maximum, mean, and standard deviation values for each collected statistic over the recording period.

© Copyright IBM Corp. 1994, 2004

## WLM Report Browser

Upon startup, the Report Browser displays. The browser shows a collection of reports. The type of display, which is user configurable, is based off the properties chosen to generate the report.

Report Browser menu options:

**New** Create report

**Close** Exit browser

**Open** Display a selected report

**Properties** Allow the properties of a report to be viewed and edited

**Delete** Delete a selected report

## Report Properties Panel

The Report Properties Panel allows the user to define the attributes that control the actual graphical representation of the WLM data. There are three tabbed panes in this panel:

- General Menu
- Tier/Class Menu
- Advanced Menu

**Report Name** A user-editable field for naming the report. Reports should end with the **.rpt** extension

## General Menu

The first tabbed pane allows the user to edit the general properties of a display as follows:

**Trend Box** Indicates that a trend report of the selected type will be generated. Trend reports allow the comparison of two different time periods on the same display. Selecting this box enables the ″End of first interval″ field for editing.

**Resource** Allows selections for the WLM resources to be displayed (such as CPU or memory). Refer to the WLM user's guide and documentation for information about the resources that can be managed.

**Width of interval**
Represents the period of time covered by any display type measuring either from the latest values available in the recording, or from user-input time selections. Interval widths are selected from this menu. The available selections vary, depending upon the tool being used:

**wlmmon** Multiple selections for minutes and hours

**wlmperf** Multiple selections for minutes, hours, days, weeks, and months

**End of first interval**
Represents the end time of a period of interest for generating a trend report. The first interval always represents a time period ending earlier than the last interval. This field can only be edited if the Trend Box is selected.

**End of last interval**
Represents the end time of a period of interest for trend and non-trend reports The last interval always represents the latest time frame to be used in generating a display report. There are two exclusive selection options for this field:

**Latest** Uses the latest time available in the recording as the end time for the report.

**Selected Time**
Allows the user to input the end time of the last interval.

## Tier/Class Menu

The second tabbed pane allows users to define the set of WLM tiers and classes to be included in a report.

**Scope** Allows the user to select a tier or class-based scope for the display. This display will vary, as tier and class concepts vary between the AIX releases (AIX 4.3 classes versus AIX 5.1 superclass and subclass definitions).

**Selection** Allows selection of including and excluding the WLM tiers or classes available in the recording.

## Advanced Menu

The third tabbed pane of the Report Properties Panel provides advanced options, primarily for the snapshot display. For snapshots, exclusive methods for coloring the display are provided for user selection. Option 1 ignores the minimum and maximum settings defined in the configuration of the WLM environment. Option 2 uses the minimum and maximum settings.

# Report Displays

There are three types of report displays:

- Snapshot Display
- Detailed Display
- Tabulation Display

Each of these displays has the following common elements:

**WLM Console**
Selections for printing or closing the display.

**Time Period** Displays the time period defined in the Report Properties Panel. For trend reports comparing two time periods, two time displays are shown.

**Tier Column** Displays the tier number associated with a class. For AIX 5.1, the column has two entries, for superclass tier (left) and subclass tier (right).

**Class Column**
Displays the class name.

**Resource Columns**
Displays the resource information based off of the type of graphical report selection chosen. These are described below.

**Status Area** Displays a set of global system performance metrics that are also recorded to aid in analysis. The set displayed may vary between AIX releases, but will include metrics such as run, queue, swap queue, and CPU busy.

## Snapshot Display

This display is a quick ″Am I OK?″ overview. The display focuses on showing class resource relationships based off user-specified variation from the defined target shares. To select or adjust the variation parameters for this display, use the Report Properties Panel Advanced Menu.

If the snapshot display is trended, the earlier (first) analysis period is shown by an arrow pointing from the earlier measurement to the later (second) measurement. If there has been no change between the periods, no arrow is shown.

## Detailed Display

In this display, the resource columns are displayed in bar-graph style, along with the percentage of measured resource activity over the time period specified. The percentage is calculated based off the total system resources defined by the WLM subsystem. If the detailed display is trended, the later (second) measurement is shown above the earlier (first) measurement interval.

## Tabulation Display

The third type of display report is a tabulation report. In this report, the following fields are provided:

**Number Sampled**
Number of recorded samples for this period

**Share Value**
Computed share value target by WLM

**Mean Value**
Calculated average over the sample period

**Standard Deviation**
Computed standard deviation

**Defined Min**
Class minimum defined in WLM limits

**Observed Min**
Actual observed minimum across time period

**Defined Soft Max**
Class soft maximum defined in WLM limits

**Defined Hard Max**
Class hard maximum defined in WLM limits

**Observed Max**
Actual observed minimum across time period

# Daemon Recording and Configuration

The daemons create recordings in the **/etc/perf/wlm** directory. For the base AIX tool **wlmmon**, these recordings are limited to the last 24–hour period.

For the Performance Toolbox tool **wlmperf**, these recordings are limited to 1 year. For the PTX, the **xmtrend** daemon is used, and uses a configuration file for recording preferences. A sample of this configuration file for WLM— related recordings is located at **/usr/lpp/perfagent.server/xmtrend_wlm.cf**. Recording customization, startup, and operation are the same as those described for the **xmtrend** daemon in Chapter 14, “Recording Performance Data on Remote and Local Systems,” on page 167.

For the base AIX, the **xmwlm** daemon is used and cannot be customized.

For recordings to be created, adequate disk allocations must be made for the **/etc/perf/wlm** directory, allowing at least 10 MB of disk space. Additionally, the daemon should be started from an **/etc/inittab** entry so that recordings can automatically restart after system reboots. The daemon will operate whether the WLM subsystem is in active, passive, or disabled (off) modes. However, recording activity is limited when WLM is off.

# Files

**/usr/bin/wlmmon**
base AIX

**/usr/bin/xmwlm**
base AIX

**/usr/bin/wlmperf**
Performance Toolbox

**/usr/lpp/perfagent.server/xmtrend.cf**
Performance Toolbox

**wlmmon and xmwlm**
Located in the **perfagent.tools** fileset.

**wlmperf and xmtrend**
Available only with the Performance Toolbox product media.

## Prerequisite

Java 1.3

**perfagent.tools**

## Exit Status

A warning message is issued by the tool if no WLM recordings are located.

## Related Information

The **wlmstat**, **wlmcntrl**, and **topas** commands.

# Chapter 13. Monitoring Remote Systems

The Performance Toolbox for AIX Agent component is a collection of programs that make it possible for a host to act as a provider of performance statistics across a network or locally. The key program is the daemon **xmservd**. This chapter and Chapter 14, “Recording Performance Data on Remote and Local Systems,” on page 167 and Chapter 15, “SNMP Multiplex Interface,” on page 177 describe the features of **xmservd**. Chapter 16, “Data Reduction and Alarms with filtd,” on page 183 describes the other important daemon in the Agent component, **filtd**. Finally, Chapter 18, “System Performance Measurement Interface Programming Guide,” on page 201 describes the local API provided with the Agent.

The remainder of this chapter first explains important features of the System Performance Measurement Interface (SPMI), which is the mechanism that provides statistics to **xmservd**, and then explains in detail how monitoring of remote systems is made possible. For that discussion, the term *data-supplier host* was adopted to describe a host that supplies statistics to another host across a network, while a host receiving the statistics over the network, processing, and displaying them is called a *data-consumer host*.

## The System Performance Measurement Interface

Monitoring statistics supplied by **xmservd** is made possible through an API called System Performance Measurement Interface (SPMI). Through the SPMI, an application can access statistics available on the local system. This is done by defining sets of statistics (statsets). Observations are taken for all the statistics of a statset at the same time. The concept of statsets is key to understanding how statistics are monitored. It is explained in the section entitled “Statsets.”

The SPMI makes extensive use of shared memory. Similarly, any dynamic data-supplier programs that extend the set of provided statistics use shared memory to export their data. The **xmservd** daemon (and properly written dynamic data-supplier programs) allocates and frees shared memory segments when starting and terminating. Some important things to know about the use of shared memory are explained in “Shared Memory Types” on page 154 and subsequent sections.

### Statsets

Each system provides a range of statistics, some of which are fixed while others, such as process statistics come and go over time. Most monitoring tasks involve the monitoring of more than one of the statistics provided by a system. In the simplest possible way to access the statistics, the requestor would ask for observations for each statistic and would issue a series of requests to get multiple statistics. This would create a number of inconveniences:

- The observations would not be taken at the same point in time.
- The overhead involved would be proportional to the number of statistics the requestor wanted.
- There would be no architected way to keep track of delta values for observations, except by the requesting program itself.

All of these inconveniences are eliminated by the definition of statsets as implemented in the SPMI. Statsets represent views of the entire data repository of statistics and are implemented as data structures that are used to keep track of *delta values* (difference between the latest observation and the previous one) for statistics. The only way an application program can read observations is by defining a statset and then requesting a reading for all the statistics in the statset. Because statsets are defined to the SPMI, which permits access to local statistics, all statistics in a statset must come from the same system.

The concept of statsets applies throughout Performance Toolbox for AIX. The program **xmperf** uses statsets to define instruments. There’s always a one-to-one relationship between an **xmperf** instrument and a statset. Similarly, every right side column of a **3dmon** graph corresponds to a statset.

© Copyright IBM Corp. 1994, 2004

Statsets are also closely related to the data packets that carry observations over the network. The **xmservd** daemon supplies data across the network in the form of data packets that correspond to statsets. Each data packet contains a time stamp that shows when a set of observations was taken and the elapsed time since the previous observation. It then contains two fields for each of the statistics in the statset. The first gives the delta value. The second contains the actual observation value.

In recording files, value records are used to carry observations. They have the same contents as the data packets and maintain the concept of statsets. When recordings are played back with **xmperf**, the statsets are used to define instruments. When a recording file is analyzed by **azizo**, statsets are not important but they are preserved when writing a filtered recording file, if requested.

## Shared Memory Types

Two types of shared memory are used by the daemon and dynamic data-supplier programs. The first is called common shared memory and is memory that all running dynamic data-supplier and local data-consumer programs (data-consumer programs that do not use the Remote Statistics Interface API) share with the daemon. The second type is allocated in one copy for each dynamic data-supplier program and is supposed to be deallocated and removed by that dynamic data-supplier program when the program exits. This second type of shared memory is called DDS shared memory.

## Common Shared Memory

The common shared memory is allocated by the SPMI library on behalf of whichever local data-consumer or data-supplier program (including **xmservd**) starts first. Each additional such program detects the common shared memory and uses the allocated segment. A counter in the common shared memory segment is incremented by one for each starting data-supplier or local data-consumer program and is decremented by one whenever one of the running programs terminate. When the counter reaches zero, the common shared memory segment is released.

Properly written data-supplier and local data-consumer programs issue a subroutine call when they terminate. This call detaches from the common shared memory segment and decrements the counter. To be properly written, the programs must detect various signals, which indicate that the program (process) is about to terminate. When one of the signals is received the program must issue the subroutine call. The call must also be issued when the program terminates in the usual way.

Most signals can be detected by a program, but some cannot. If one of the undetectable signals causes the program (process) to terminate, the subroutine call is not issued. As a result, the common shared memory segment never is released, since the counter never reaches zero. If this happens, you must release the common shared memory manually, as described in “Releasing Shared Memory Manually” on page 155.

To avoid the situation, never stop a data-supplier or local data-consumer program with the option **-9**. That would end the program with a **SIGKILL** signal, which is not detectable.

## DDS Shared Memory

A dynamic data-supplier program exports its data through a private shared memory area, called the DDS shared memory area, which is allocated by the SPMI API. If the memory area exists when the dynamic data-supplier program starts, the program stops. This ensures that the same dynamic data-supplier program is not running in multiple copies. It also places the responsibility for releasing the DDS shared memory segment whenever the dynamic data-supplier program stops on the program itself.

Properly designed dynamic data-supplier programs detect signals, which cause the program to stop and issue a subroutine call to release shared memory. One single subroutine call is used to release DDS shared memory and disassociate the program from the common shared memory.

If a dynamic data-supplier program is stopped in a way that cannot be detected from the program itself, the DDS shared memory is not released and subsequent attempts to start the dynamic data-supplier program fail. If this happens, you must release the DDS shared memory manually, as described in section entitled “Releasing Shared Memory Manually.”

To avoid the situation, never kill a dynamic data-supplier program with the option **-9**. That would stop the program with a **SIGKILL** signal, which is not detectable.

## Releasing Shared Memory Manually

In situations where one or more data-supplier or local data-consumer programs have stopped in such a way that their shared memory allocations have not been released, the shared memory segments should be released from the command line before attempting to restart the programs. It is recommended that all data-supplier and local data-consumer programs, including **xmservd**, are killed before you attempt to release shared memory. Clearing of all shared memory segments could be done through the following steps.

1. Identify all data-supplier and local data-consumer programs that are running. Use the **ps** command and your knowledge of the programs in use on your system to locate all of them. For each of the running data-supplier or local data-consumer programs, note their process IDs.
2. Stop all processes associated with data-supplier and local data-consumer programs without using a command line flag.
3. Verify that all data-supplier and local data-consumer processes have been stopped. If not, use the **kill -9** command to stop them.
4. List the shared memory segments in use with the command **ipcs -m**. This produces a list like the following:

```
IPC status from /dev/mem as of
Fri Dec 31 07:54:44 CST 1993
T  ID     KEY        MODE        OWNER  GROUP
Shared Memory:
m      0  0x0d050296 --rw-------   root system
m  20481  0x5806188b --rw-rw-rw- nchris system
m  28674  0x780502ea --rw-rw-rw-   root system
m  12292  0x780502e3 --rw-rw-rw-   root system
m  20485  0x780502d1 --rw-rw-rw-   root system
```

5. Identify all shared memory segments with a KEY that begins with “0x78”. All shared memory allocated by data-supplier and local data-consumer programs has this key. Now use the **ipcrm** command to remove the shared memory segments, specifying as command arguments the IDs of the segments you want to remove. To remove all three data-supplier and local data-consumer segments listed above, your command would be:

```
ipcrm -m 28674 -m 12292 -m 20485
```

6. Restart the dynamic data-supplier and data-consumer programs as required. To start **xmservd** and any dynamic data-supplier programs started by it, simply execute the command **xmpeek**.

## The xmservd Command Line

The **xmservd** daemon is always started from **inetd**. Therefore, command line options must be specified on the line defining **xmservd** to **inetd** in the file **/etc/inetd.conf**. The general format of the command line is:

**xmservd** [**-v**] [**-b** *UDP_buffer_size*] [**-i** *min_remote_interval*] [**-l** *remove_consumer_timeout*] [**-m** *supplier_timeout*] [**-p** *trace_level*] [**-s** *max_logfile_size*] [**-t** *keep_alive_limit*] [**-x** *xmservd_execution_priority*]

All command line options are optional. The options are:

**-v**
: Verbose. Causes parsing information for the **xmservd** recording configuration file to be written to the **xmservd** log file.

**-b** Defines the size of the buffer used by the daemon to send and receive UDP packets. The buffer size must be specified in bytes and can be from 4,096 to 16,384 bytes. The buffer size determines the maximum number of data values that can be sent in one **data_feed** packet. The default buffer size is 4096 bytes, which allows for up to 124 data values in one packet.

**-i** Defines the minimum interval in milliseconds with which data feeds can be sent. Default is 500 milliseconds. A value between 100 and 5,000 milliseconds can be specified. Any value specified is rounded to a multiple of 100 milliseconds. Whichever minimum remote interval is specified causes all requests for data feeds to be rounded to a multiple of this value. See further details in "Rounding of Sampling Interval."

**-l** (Lowercase L). Sets the **time_to_live** after feeding of statistics data has ceased as described in "Life and Death of xmservd" on page 157. Must be followed by a number of minutes. A value of 0 (zero) minutes causes the daemon to stay alive forever. The default **time_to_live** is 15 minutes.

This value is also used to control when to remove inactive data-consumers as described in "Removing Inactive Data Consumers" on page 158.

**-m** When a dynamic data-supplier is active, this value sets the number of seconds of inactivity from the DDS before the SPMI assumes the DDS is dead. When the timeout value is exceeded, the **SiShGoAway** flag is set in the shared memory area and the SPMI disconnects from the area. If this flag is not given, the timeout period is set to 90 seconds.

The size of the timeout period is kept in the SPMI common shared memory area. The value stored is the maximum value requested by any data consumer program, including **xmservd**.

**-p** Sets the trace level, which determines the types of events written to the log file **/etc/perf/xmservd.log1** or **/etc/perf/xmservd.log2**. Must be followed by a digit from 0 to 9, with 9 being the most detailed trace level. Default trace level is 0 (zero), which disables tracing and logging of events but logs error messages.

**-s** Specifies the approximate maximum size of the log files. At least every **time_to_live** minutes, it is checked if the currently active log file is bigger than **max_logfile_size**. If so, the current log file is closed and logging continues to the alternate log file, which is first reset to zero length. The two log files are **/etc/perf/xmservd.log1** and **/etc/perf/xmservd.log2**. Default maximum file size is 100,000 bytes. You cannot make **max_logfile_size** smaller than 5,000 or larger than 10,000,000 bytes.

**-t** Sets the **keep_alive_limit** described in section "Life and Death of xmservd" on page 157. Must be followed by a number of seconds from 60 to 900 (1 to 15 minutes). Default is 300 seconds (5 minutes).

**-x** Sets the execution priority of **xmservd**. Use this option if the default execution priority of **xmservd** is unsuitable in your environment. Generally, the daemon should be given as high execution priority as possible (a smaller number gives a higher execution priority).

## Rounding of Sampling Interval

As explained under the **-i** command line argument, all sampling intervals requested by remote data-consumer programs are rounded to the effective minimum sampling interval of **xmservd**. This can cause unintended rounding of sampling intervals as shown in the rounding of sampling interval by **xmservd**. This rounding can be eliminated by always using 100 milliseconds as the minimum sampling interval. However, if you use 100 milliseconds, and remote data-consumer programs use a wide variety of sampling intervals, then the overhead of **xmservd** increases because it has to set its interval timer to do processing more frequently. Generally, the minimum sampling interval should be set to as large a value as possible, preferably 1000 milliseconds or more.

The following example illustrates rounding of sampling interval for different minimum sampling intervals and various requested sampling intervals:

```
minimum remote interval  requested interval  resulting interval
-----------------------  ------------------  ------------------
        200                    500             600
        200                    3,000           3,000
        200                    1,000           1,000

        300                    500             600
        300                    3,000           3,000
        300                    1,000           900

        400                    500             400
        400                    3,000           3,200
        400                    1,000           1,200

        500                    500             500
        500                    3,000           3,000
        500                    1,000           1,000
```

## The xmservd Interface

The **xmservd** daemon is designed to be started from the **inetd** "super daemon." Even when you start the daemon manually it reschedules itself via **inetd** and lets the manually started process die. The following sections describe how **xmservd** starts, terminates, and keeps track of data-consumer programs.

## Life and Death of xmservd

The **xmservd** daemon must be configured as an **inetd** daemon to run properly. If you do start the daemon manually, it attempts to reschedule itself by invoking the program **xmpeek** and then exit. This causes **xmservd** to be rescheduled via **inetd**. The line defining the daemon in **/etc/inetd.conf** must specify the "wait" option to prevent inetd from starting more than one copy of the daemon at a time. The file **/etc/inetd.conf** is prepared during the installation of the Agent component.

If you want the daemon to be started automatically as part of the boot process, you can add the following two lines at the very end of the file **/etc/rc.tcpip**:

```
/usr/bin/sleep 10
/usr/bin/xmpeek
```

The first line is necessary only when you intend to use the xmservd/SMUX interface to export statistics to the local SNMP agent.

**Note:** The xmservd/SMUX interface is only available on **RS/6000** Agents.

The Chapter 15, "SNMP Multiplex Interface," on page 177 describes the xmservd/SMUX interface. The line with the sleep command makes sure the start of the **snmpd** daemon is completed before **xmservd** starts. The second line uses the program **xmpeek** (described later in this chapter) to kick off the **xmservd** daemon.

The **xmservd** daemon is started by **inetd** immediately after a UDP datagram is received on its port. Note that the daemon is not scheduled by a request through the SMUX interface from the local SNMP agent. This is because the SNMP agent uses a different port number. Unless **xmservd** ends abnormally or is killed, it continues to run as long as any data-consumer needs its input or a connection to the SNMP agent is established and alive. When no data-consumer needs its input and either no connection was established through the SMUX interface or any such connection is terminated, the daemon hangs around for **time_to_live** minutes as specified with the **-l** (lowercase L) command line argument to **xmservd**. The default number of **time_to_live** minutes is 15.

In some environments, it make take some time for a system's **xmservd** daemon to respond to invitations from remote data-consumers. This can be because the network route is long or the network congested; it may be because all memory on the system is in use so pages must be paged out before **xmservd** can be

loaded; it may be because the **xmservd** executable is loaded off a server, as on diskless systems. In either case, the data-consumer program may not receive a response from the system in time. Most remote data-consumer programs in Performance Toolbox for AIX have ways to extend the time they wait for responses. See the command lines for each data-consumer program for specifics.

Whenever a connection to the SNMP agent through the SMUX interface is active, or whenever **xmservd** is configured to record performance data to a file (see Chapter 14, "Recording Performance Data on Remote and Local Systems," on page 167) the daemon does not time out and die even when there are no data-consumers to supply. In these situations, the **time_to_live** limit is used only to determine when to look for inactive remote consumers that can be deleted from the tables in **xmservd**.

## Signals Understood by xmservd

Like many other daemons, **xmservd** interprets the receipt of the signal **SIGHUP** (kill -1) as a request to refresh itself. It does this by spawning another copy of itself via **inetd** and kill itself. When this happens, the spawned copy of **xmservd** is initially unaware of any data consumers that may have been using the copy of **xmservd** that received the signal. Consequently, all data-consumer programs must request a resynchronizing with the spawned daemon to continue their monitoring.

The other signal recognized by **xmservd** is **SIGINT** (kill -2) that causes the daemon to dump any MIB data it has to a file as described in the section Interaction Between xmserv and SNMP ("Interaction Between xmservd and SNMP" on page 178).

## Removing Inactive Data Consumers

When a data-consumer program such as **xmperf** uses broadcasts to contact data-supplier hosts, most likely the monitor defines instruments (each of which causes **xmservd** to create a statset) with only a few of the daemons that respond. Consequently, most daemons have been contacted by many data consumers but supply statistics to only a few. This causes the host tables in the daemon to swell and, in the case of large installations, can induce unnecessary load on the daemon. To cope with this, the daemon attempts to get rid of data consumers that appear not to be interested in its service.

The **time_to_live** parameter is used to check for inactive partners. A data consumer is removed from the daemon's tables if either of the following conditions is true:

1. No packet was received from the data consumer for twice the **time_to_live** period and no statsets were defined for the data consumer.
2. No packet was received from the data consumer for eight times the **time_to_live** period and none of the defined statsets are feeding data to the data consumer.

A data consumer that is subscribing to **except_rec** messages is treated as if it had a statset defined with the daemon.

## Checking that Data Consumers are Alive

When **xmservd** is running and supplying input to one or more data consumers, it must make sure that the data consumers are still alive and needing its input. If not, it would be a waste of system resources to continue sending statistics across the network. The daemon uses a **keep_alive_limit** to determine when it's time to check that data-consumer hosts are still alive. The alive limit is reset whenever the user makes changes to the remote monitoring configuration from the data-consumer host, but not when data is fed to the data consumer.

When the **keep_alive_limit** is reached, **xmservd** sends a message of type **still_alive** to the data consumer. The data-consumer program has **keep_alive_limit** seconds to respond. If a response is not received after **keep_alive_limit** seconds, the daemon sends another **still_alive** message and waits another **keep_alive_limit** seconds. If there's still no response, the daemon assumes the data consumer to be dead or no longer interested and stops sending statistics to it. The default **keep_alive_limit** is 300 seconds (five minutes); it can be set with the **-t** command line argument to **xmservd**.

## Handling Exceptions

Through the program **filtd** described in Data Reductions and Alarms with **filtd** (Chapter 16, “Data Reduction and Alarms with filtd,” on page 183), you can define exception conditions that can cause one or more actions to be taken. One such action is the execution of a command on the host where the daemon runs; another is the sending of an exception message. The message type **except_rec** is used for the latter.

The contents of each exception message is:

1. The host name of the host sending the exception message.
2. The time when the exception was detected.
3. The severity of the exception, a number between 0 and 10.
4. The minimum number of minutes between two exception messages from a given exception definition.
5. A symbolic name describing the exception.
6. A more verbose description of the exception.

The **xmservd** daemon sends exceptions to all hosts it knows that have declared that they want to receive exception messages. The **RSiOpen** and **RSiInvite** subroutine calls of the API are used by the data-consumer application to declare whether it wants to receive exception messages.

The program **exmon** is especially designed to monitor exception messages. It allows its user to specify which hosts to monitor for exceptions and displays a window with a matrix that shows which hosts generated exceptions, what types were generated, and how many of each type. This program is described in Chapter 8, “Monitoring Exceptions with exmon,” on page 85.

Currently, **xmperf** does not request exception messages unless you set the X resource **GetExceptions** to true or use the **-x** command line argument. If you have requested exceptions this way and one is received by **xmperf**, it is sent to the **xmperf** main window where it appears as a text message. No other action is taken by **xmperf**.

## Session Recovery by xmservd

If the **xmservd** daemon dies or is killed while one or more data consumers have statsets defined with it, the daemon attempts to record the connections in the file **/etc/perf/xmservd.state**. If this file exists when **xmservd** later is restarted, a message of type **i_am_back** is sent to each of the data-consumer hosts recorded in the file. The file is then erased.

If the programs acting as data consumers are capable of doing a resynchronizing, the interrupted monitoring can resume swiftly and without requiring manual intervention. The **xmperf** and **3dmon** programs can and do resynchronize all active monitors for a host whenever an **i_am_back** message is received from that host.

## The xmquery Network Protocol

Several types of messages (packets) that flow between data-supplier hosts and data-consumer hosts were previously mentioned. Message types are organized in four groups as follows:

```
Configuration Messages
        create_stat_set    Type = 01
        del_set_stat       Type = 02
        first_cx           Type = 03
        first_stat         Type = 04
        instantiate        Type = 05
        next_cx            Type = 06
        next_stat          Type = 07
        path_add_set_stat  Type = 08
        path_get_cx        Type = 09
        path_get_stat      Type = 10
```

```
        stat_get_path      Type = 11

Data Feed and Feed Control Messages
        begin_feeding      Type = 31
        change_feeding     Type = 32
        end_feeding        Type = 33
        data_feed          Type = 34
        going_down         Type = 35

Session Control Messages
        are_you_there      Type = 51
        still_alive        Type = 52
        i_am_back          Type = 53
        except_rec         Type = 54

Status Messages
        send_status        Type = 81
        host_status        Type = 82
```

## Configuration Messages

All the configuration messages are specific to the negotiation between the data consumer and the data supplier about what statistics should be sent by the data supplier. Note that all such messages require a response, and that they all are initiated by the data consumer.

## Data Feed and Data Feed Control Messages

When the negotiation of what data to supply is completed, the data-supplier host's **xmservd** maintains a set of information about the statistics to supply. A separate set is kept for each data-consumer program. No feeding of data is started until a **begin_feeding** message is received from the data-consumer program. The **begin_feeding** message includes information about the frequency of data feeds and causes **xmservd** to start feeding data at that frequency, using **data_feed** packets.

Data feed to a data consumer continues until that data consumer sends an **end_feeding** message or until the data consumer does no longer respond to **still_alive** messages. At that time data feeding stops.

The frequency of data feeds can be changed by the data-consumer program by sending the **change_feeding** message. This message is sent whenever the user changes the interval property of an **xmperf** instrument.

The final message type in this group is **going_down**. This message is sent by **xmperf** and the other remote data-consumer programs in Performance Toolbox for AIX whenever they terminate orderly and whenever any other program written to the RSi API (see Chapter 19, "Remote Statistics Interface Programming Guide," on page 245) issues the **RSiClose** call. The message is sent to all data-supplier hosts that the data-consumer program knows about (or the host **RSiClose** is issued against) and causes the daemons on the data-supplier hosts to erase all information about the terminating data-consumer program.

## Session Control Messages

Two of the session control message types have already been mentioned in previous sections. To recapture, **are_you_there** is sent from a data consumer to provoke potential data-supplier hosts to identify themselves. The **still_alive** message is the only message type that is initiated by **xmservd** without input from a data consumer. It prompts remote monitors to respond and thus prove that they are still alive.

The third session control message is the **i_am_back** message, which is always the response to the first message **xmservd** receives from a data consumer.

### Resynchronizing in xmperf

When an **i_am_back** message is received by a data-consumer host's **xmperf** program, it responds by marking the configuration tables for the data-supplier host as void. This is because the data-supplier host's **xmservd** daemon has obviously restarted, which means that earlier negotiations about statsets are now invalidated.

If an **i_am_back** message is received from a remote supplier while an instrument for that supplier is active, a renegotiation for that instrument is started immediately. If other remote instruments for the supplier are defined to the data-consumer host, renegotiation for those instruments is delayed until the time each instrument is activated.

Renegotiation is not started unless **xmperf** on the data-consumer host takes action. It is quite possible that a data-supplier host is rebooted and its **xmservd** daemon therefore goes quietly away. The data consumer no longer receives data, and the remote instruments stop playing. Currently, no facility detects this situation but a menu option allows the user to resynchronize with a data supplier. When this option is chosen, an **are_you_there** message is sent from the **xmperf**. If the data-supplier daemon is running or can be started, it responds with an **i_am_back** message and renegotiation starts.

## Status Messages and the xmpeek Program

If a large number of data-consumer programs each is monitoring several statistics from one single data-supplier host, the sheer number of requests that must be processed can result in more load on the data-supplier host than is feasible.

Two features allow you to control the daemon on any host you are responsible for. The first one is a facility to display the status of a daemon, as described in this section. The other is the ability to control the access to the **xmservd** daemon as described in "Limiting Access to Data Suppliers" on page 164.

Because the **xmservd** daemon runs in the background and may start and stop as required, special action is needed to determine the status of the daemon. Such action is implemented through the two message types **send_status** and **host_status**. The first can be sent to any **xmservd** daemon, which then responds by returning the message with total counts for the daemon's activity, followed by a message of type **host_status** for each data consumer it knows.

A program called **xmpeek** is supplied as part of the Performance Toolbox for AIX. This program allows you to ask any host about the status of its **xmservd** daemon. The command line is simple:

**xmpeek** [**-a**|**-l**] [*hostname*]

Both flags are optional. The **-l** flag (lowercase L) is explained in "Using the xmpeek Program to Print Available Statistics" on page 163. If the flag **-a** is specified, one line is listed for each data consumer known by the daemon. If omitted, only data consumers that currently have instruments (statsets) defined with the daemon are listed.

If a host name is specified, the daemon on the named host is asked. If no host name is specified, the daemon on the local host is asked. The following is an example of the output from the **xmpeek** program:

```
Statistics for xmservd daemon on *** birte ***
Instruments currently defined:    1
Instruments currently active:     1
Remote monitors currently known:  2
--Instruments--- Values  Packets
Defined  Active  Active  Sent
Internet Address  Port  Hostname
------- ------- ------- ------- ----------------  ----  --------
1       1       16       3,344   129.49.115.208   3885  xtra
```

Output from **xmpeek** can take two forms.

The first form is a line that informs you that the **xmservd** daemon is not feeding any data-consumer programs. This form is used if no statsets are defined with the daemon and no command flags are supplied.

The second form includes at least as much as is shown in the Sample Output from xmpeek (“Status Messages and the xmpeek Program” on page 161), except that the single detail line for the data consumer on host **xtra** only is shown if either the **-a** flag is used or if the data consumer has at least one instrument (statset) defined with the daemon. Note that **xmpeek** itself appears as a data consumer because it uses the RSi API to contact the daemon. Therefore, the output always shows at least one known monitor.

In the fixed output, first the name of the host where the daemon is running is shown. Then follows three lines giving the totals for current status of the daemon. In the above example, you can see that only one instrument is defined and that it’s active. You can also see that two data consumers are known by the daemon, but that only one of them has an instrument defined with the daemon in **birte**. Obviously, this output was produced without the **-a** flag.

An example of more activity is shown in the following example output from xmpeek. The output is produced with the command:

```
xmpeek -a birte
```

**Note:** Some detail lines show zero instruments defined. Such lines indicate that an **are_you_there** message was received from the data consumer but that no states were ever defined or that any previously defined states were erased.

```
Statistics for smeared daemon on *** birte ***
   Instruments currently defined:    16
   Instruments currently active:     14
   Remote monitors currently known:   6
--Instruments--- Values Packets  Internet Protocol
 Defined Active  Active  Sent        Address       Port  Hostname
 ------- ------  ------ ------   ----------------  ----  --------
 8       8       35     10,232   129.49.115.203    4184   birte
 6       4       28      8,322   129.49.246.14     3211   umbra
 0       0        0          0   129.49.115.208    3861   xtra
 1       1       16      3,332   129.49.246.14     3219   umbra
 0       0        0          0   129.49.115.203    4209   birte
 1       1       16        422   129.49.115.208    3874   xtra
------- ------- ------- ------
 16      14      95     22,308
```

Notice that the same host name may appear more than once. This is because every running copy of **xmperf** and every other active data-consumer program is counted and treated as a separate data consumer, each identified by the port number used for UDP packets as shown in the **xmpeek** output.

The second detail line in the Sample Output from **xmpeek** (“Status Messages and the xmpeek Program” on page 161) shows that one particular monitor on host **umbra** has six instruments defined but only four active. This would happen if a remote **xmperf** console has been opened but is now closed. When you close an **xmperf** console, it stays in the Monitor menu of the **xmperf** main window and the definition of the instruments of that console remains in the tables of the data-supplier daemon but the instruments are not active.

## Instrument Status in xmperf

If the data-consumer program is **xmperf**, there are only three ways an instrument can be erased from the tables in the **xmservd** daemon after it is defined. They are:

1. You can erase an instrument in a remote console or in an instantiated remote skeleton console.
2. You can erase a remote console or an instantiated remote skeleton console.
3. The daemon takes the initiative to erase its information about an instrument after it has detected that the data consumer, which defined the instrument is no longer active.

In most cases, the latter situation occurs because the data consumer has been killed (as opposed to closed down orderly). As the daemon detects that the instruments of the data-consumer hosts are no longer active, it deletes them one at a time. When the last instrument of a data consumer is deleted from the tables in **xmservd**, all information about the remote monitor is deleted too, and the monitor no longer shows up in the output from **xmpeek**.

## Using the xmpeek Program to Print Available Statistics

If the **xmpeek** program is invoked with the **-l** flag (lowercase L) it lists all the available statistics of the remote host given on the command line, or the local host if no host name is given. The list of statistics is sent to standard output, which permits you to redirect it to a file or pipe it into another command. The following figure shows a partial listing of statistics on an HP 9000/7255:

```
/hp2/CPU/                Central processor statistics
/hp2/CPU/gluser          System-wide time executing in user mode (percent)
/hp2/CPU/glkern          System-wide time executing in kernel mode (percent)
/hp2/CPU/glwait          System-wide time waiting for IO (percent)
/hp2/CPU/glidle          System-wide time CPU is idle (percent)
/hp2/CPU/glnice          System-wide time CPU is running w/nice priority (%)

  . . .

/hp2/CPU/cpu0/           Statistics for processor #0
/hp2/CPU/cpu0/user       Time executing in user mode (percent)
/hp2/CPU/cpu0/kern       Time executing in kernel mode (percent)
/hp2/CPU/cpu0/wait       Time waiting for IO (percent)
/hp2/CPU/cpu0/idle       Time CPU is idle (percent)
/hp2/CPU/cpu0/nice       Time CPU is running code with nice priority

  . . .

/hp2/Mem/                Memory Statistics
/hp2/Mem/Real/           Physical memory statistics
/hp2/Mem/Real/size       Size of physical memory (4K pages)
/hp2/Mem/Real/numfrb     Number of pages on free list
/hp2/Mem/Real/%free      % memory which is free
/hp2/Mem/Real/totreal    Total real memory (Kbytes?'
/hp2/Mem/Real/actreal    Active real memory (Kbytes?'
/hp2/Mem/Virt/           Virtual memory management statistics
/hp2/Mem/Virt/pagein     4K pages read by VMM
/hp2/Mem/Virt/pageout    4K pages written by VMM
/hp2/Mem/Virt/zerofill   Page faults satisfied by zero-filling memory frames
/hp2/Mem/Virt/pagexct    Total page faults

 . . .
```

When a host's statistics include contexts that may exist in multiple instantiations and such instantiations are volatile, the list does not break all such contexts down in their components. Rather, only the first instance of the context is broken down and all further instances are listed with five dots appended to the statistics path name. The following example shows this. The process identified by **514~wait** (actually a pseudo process) is fully broken down. All other processes are merely listed with their identifier since they would all break down to the same base statistics as the wait process.

```
/birte/Proc/                  Process statistics
/birte/Proc/pswitch           Process context switches
/birte/Proc/runque            Average count of processes waiting for the CPU
/birte/Proc/runocc            Number of samplings of runque
/birte/Proc/swpque            Average count of processes waiting to be paged in
/birte/Proc/swpocc            Number of samplings of swpque
/birte/Proc/ksched            Number of kernel process creations
/birte/Proc/kexit             Number of kernel process exits
/birte/Proc/514~wait/         Process wait  (514) %cpu 54.6, PgSp: 0.0mb, uid:
/birte/Proc/514~wait/pri      Process priority
/birte/Proc/514~wait/wtype    Process wait status
/birte/Proc/514~wait/majflt   Process page faults involving IO
```

```
/birte/Proc/514~wait/minflt     Process page faults not involving IO
/birte/Proc/514~wait/cpums      CPU time in milliseconds in interval
/birte/Proc/514~wait/cpuacc     CPU time in milliseconds in life of process
/birte/Proc/514~wait/cpupct     CPU time in percent in interval
/birte/Proc/514~wait/usercpu    Process CPU use in user mode (percent)
/birte/Proc/514~wait/kerncpu    Process CPU use in kernel mode (percent)
/birte/Proc/514~wait/workmem    Physical memory used by process private data (4K)
/birte/Proc/514~wait/codemem    Physical memory used by process code (4K pages)
/birte/Proc/514~wait/pagsp      Page space used by process private data (4K page
/birte/Proc/514~wait/nsignals   Signals received by process
/birte/Proc/514~wait/nvcsw      Voluntary context switches by process
/birte/Proc/514~wait/tsize      Code size (bytes)
/birte/Proc/514~wait/maxrss     Maximum code+data resident set size (4K pages)
/birte/Proc/12002~x/.....
/birte/Proc/13207~xlock/.....
/birte/Proc/771~netw/.....
/birte/Proc/1~init/.....
/birte/Proc/5723~trapgend/.....
/birte/Proc/0~/.....
/birte/Proc/15339~aixterm/.....
/birte/Proc/2823~syncd/.....
/birte/Proc/13047~xmservd/.....
/birte/Proc/15593~aixterm/.....

 . . .
```

## Protocol Version Control

Because the Performance Toolbox for AIX can be expanded in the future, it is likely that changes to messages or network protocol will be introduced. For this reason, the message types **are_you_there**, **i_am_back**, and **send_status** carry information about the **xmquery** protocol level they are using.

In case of a difference in protocol version, data-consumer programs do not attempt to negotiate with the data-supplier host. This does not prevent the data supplier from negotiating with, and supplying data to, other remote monitors at the same protocol level as itself.

## Limiting Access to Data Suppliers

Access to the **xmservd** daemon can be limited by supplying stanzas in the configuration file **/etc/perf/xmservd.res** (or **/usr/lpp/perfagent/xmservd.res** if the file **/usr/lpp/perfagent/xmservd.res /etc/perf/xmservd.res** does not exist). The three stanzas follow. Note that the colon is part of the stanza. The stanza must begin in column one of a line. There may be more than one line for each stanza type, but in the case of the **max:** stanza, the last instance overrides any earlier.

**only:** When this stanza type is used, access to the daemon is restricted to hosts that are named after the stanza. Hostnames are specified separated by blanks, tabs or commas. Access from any host that is not specified in an **only:** line is rejected at the time an **are_you_there** message is received.

Be sure you understand this: If one or more **only:** lines are specified, only hosts specified in such lines get through to the data retrieval functions of the daemon.

**always:**
When this stanza type is used, access to the daemon is always granted to hosts that are named after the stanza. Hostnames are specified separated by blanks, tabs or commas. The idea is to make sure that persons who need to do remote monitoring from their hosts can indeed get through, even if the number of active data consumers exceeds the limit established.

However, if an **only:** stanza is also specified, but the host is not named in such stanza line, access is denied even before the **always:** stanza can be checked. Consequently, if you use the **always:** stanza, you must either refrain from using the **only:** stanza or make sure that all hosts named in the **always:** lines are also named in the **only:** lines.

**max:** This stanza must be followed by the number of simultaneous data consumers that are allowed to define statsets with the daemon at any one time. Any data consumers running from hosts named in **always:** lines are not counted when it is checked if the maximum is exceeded.

Access is denied at the time a statset is defined, which usually is when a remote console is opened from the data-consumer host.

If no **max:** line is found, the maximum number of data consumers defaults to 16.

The following shows a sample **xmservd** configuration file. Two **only:** lines define a total of nine hosts that can access the **xmservd** daemon. No other host is allowed to request statistics from the daemon on the host with this configuration file.

Two **always:** lines name two hosts from where remote monitoring should always be allowed. Finally, a maximum of three data consumers at a time are permitted to have statsets defined. Note that each copy of **xmperf** and the other remote data-consumer programs of Performance Toolbox for AIX count as one data consumer, no matter on which host they run.

```
only:   srv1   srv2   birte  snavs  xtra  jones  chris
only:   savanna rhumba
always: birte
always: chris
max: 3
```

## Starting Dynamic Data-Supplier Programs

The **xmservd** daemon supplies statistics to data consumers. Such statistics may be maintained and updated internally by **xmservd** itself through the SPMI API or may be marketed by **xmservd** to data consumers on behalf of other manufacturers of statistics. Programs that provide **xmservd** with statistics in this way are called dynamic data-supplier (DDS) programs. They are written to the application programming interface of the System Performance Measurement Interface (see the System Performance Measurement Interface API (Chapter 18, “System Performance Measurement Interface Programming Guide,” on page 201)).

Before a DDS can start supplying statistics to **xmservd**, the DDS must register with **xmservd**. Before it can do this, it must be started. DDS programs can be started manually or by any other process when their presence is required, but some dynamic data suppliers may always be required to start when **xmservd** starts. To facilitate this, the **xmservd** configuration file in **/etc/perf/xmservd.res** (If the file **/etc/perf/xmservd.res** does not exist, the file **/usr/lpp/perfagent/xmservd.res** is used.) has a special type of stanza to identify DDS programs that must be started by **xmservd** whenever **xmservd** starts. The stanza can occur as many times as you have DDS programs to start, each line describing one DDS program. The stanza is:

**supplier:**
The stanza must be followed by at least one byte of white space and the full path name of the executable dynamic data-supplier program as shown in the following example:

```
supplier: /usr/samples/perfagent/server/SpmiSupl
supplier: /u/jensen/mysuppl -x -k 100
supplier: /usr/bin/filtd -p5
```

The example contains three stanzas as follows:

- One of the sample programs described in the System Performance Measurement Interface API (Chapter 18, “System Performance Measurement Interface Programming Guide,” on page 201).
- A DDS program called **mysuppl** which you may eventually write.
- The data reduction and alarm daemon **filtd** as described in the Chapter 16, “Data Reduction and Alarms with filtd,” on page 183 chapter.

Your **mysuppl** program, apparently, takes command line arguments as does the **filtd** daemon. The example also shows how these command line arguments can be put into the file.

```
supplier: /usr/samples/perfagent/server/SpmiSupl
supplier: /u/jensen/mysuppl -x -k 100
supplier: /usr/bin/filtd -p5
```

## Adjusting Socket Buffer Pool

If you use the Performance Toolbox for AIX in a network where a large number of hosts are running the **xmservd** daemon, you may have to increase the maximum size of the socket buffer pool on data consumer hosts to reduce the probability of UDP packets being dropped.

If you notice that **xmperf** does not see all the hosts that run **xmservd**, chances are that UDP drops packets. Use the **no** command to increase the socket buffer pool from four to eight times the default. For example: no -o sb_max=262144

In AIX 3.2, if packets still seem to be dropped, use the **netstat -m** command to display the “requests for memory denied.” If this number grows as you refresh the host list, use the **no** command to increase the “lowclust” option like this:

```
no -o lowclust=50
```

To make sure the values are increased each time your host boots, add the previous commands to the file **/etc/rc.tcpip**.

# Chapter 14. Recording Performance Data on Remote and Local Systems

This chapter provides information about recording performance data on remote and local systems.

## Recording on Remote and Local Systems Overview

Monitoring of performance data through the network is important and extremely useful if you know when and what to monitor. Unfortunately, that is not usually the case. Often performance problems arise and impact end users while the system administrator is unaware these problems until it is too late to start a monitoring session.

The **xmtrend** daemon and the **xmservd** daemon can be used to record system performance data, **xmtrend** and **xmservd**. Both daemons permit any system with the Agent component installed to record the activity on the system at all or selected times and for any set of performance statistics. This allows a system administrator to use the activity recording for an after-the-fact analysis of the performance problems. Both daemons are controlled through recording configuration files.

The two recording daemons are provided to address different recording and analysis philosophies. The **xmservd** agent is an existing daemon that can simultaneously provide near real-time network-based data monitoring and local recording on a given node. It focuses on high access rates for local and remote consumers like **xmperf** and **3dmon**. Local recordings created by **xmservd** are also typically high-rate on a limited set of metrics. This capability, however, is not optimized for long-term recordings. Typical **xmservd** recordings can consume several megabytes of disk storage every hour.

For this reason, the **xmtrend** agent was created to focus on providing manageable 24 x 7 long-term recordings of large metric sets. This daemon operates independently of **xmservd**. These recordings are used by **jazizo**, **jtopas** and other analysis tools supporting the trend recording format. The user specifies in a configuration file which statistics are to be recorded. The daemon then automatically computes and records the Maximum, Minimum, Mean, and Standard Deviation for each listed metric, across a frequency specified by the user. Like **xmservd**, the **xmtrend** agent uses the Spmi interface to request data at an internal cycle rate of at least once per second. For **xmtrend**, this cycle rate is independent of the recording frequency specified by the user, which by default is once per 10 minutes.

Whenever **xmservd** is configured to record the activity of the system where it is running, the daemon is prevented from dying as described in “Life and Death of xmservd” on page 157. The daemon considers itself to be configured for local recording only if a recording configuration file is present.

All recording files created by **xmservd** or **xmtrend** are placed in the **/etc/perf** directory unless otherwise specified. Recording file names are of the format **azizo.yymmdd** for **xmservd** and **xmtrend.yymmdd** for **xmtrend** where the part after the period is built from the day the first record was written to the file. A recording for February 26, 1994 would thus be called **/etc/perf/azizo.940226**. The recording activity for any one day always goes to the same file, even when **xmservd** or **xmtrend** is stopped and started over the same day. If a recording file for the day exists when **xmservd** or **xmtrend** starts, it appends additional activity to that file; otherwise it creates the file. For further details about how **xmservd** or **xmtrend** uses recording files, see the “Retain Line” on page 168 section.

For top recordings supporting the **jtopas** client, a special configuration file is used, along with the **-T** command line option. Top recordings are created in the **/etc/perf/Top** directory, and only recordings in that directory are recognized by the **jtopas** client. Recordings are made in the following format:

```
jtopas.YYMMDD
```

© Copyright IBM Corp. 1994, 2004

Recordings produced by either daemon have one or more sets of statistics. One is created for each recording interval defined in the recording configuration file. Each statset is assigned a number equal to the recording interval divided by the minimum sampling interval of the daemon.

## Recording Configuration File

The recording configuration file must be supplied by the system administrator who configures a host. No recording configuration file is supplied as part of the Performance Toolbox for AIX. The file is in ASCII format. When **xmservd** starts, it first tries to locate the recording configuration file as **/etc/perf/xmservd.cf**. If this file doesn't exist, **xmservd** looks for the recording configuration file as **/usr/lpp/perfagent/xmservd.cf**. If either file exists, **xmservd** considers itself configured for recording and parses the recording configuration file for instructions about when and what to record. The **xmtrend** agent works in a similar manner with a few exceptions.

The **xmtrend** agent looks for the **/etc/perf/xmtrend.cf** configuration file. If not found, the program exits. A sample configuration file is provided at **/usr/lpp/perfagent/xmtrend.cf**. The **xmtrend** agent does have some command line arguments to allow the user to specify where **xmtrend** should look for the configuration file, this is described later.

The recording configuration file must contain the following lines:

- One retain line
- One frequency line
- One or more metric lines
- One or more start-stop lines

The recording configuration file may also contain the following:

- One or more command lines
- One or more hot lines

## Configuration File Lines

The following sections describe the lines in the recording configuration file. They must be displayed in the sequence shown, and the keywords or metric names must begin in column one of each line. White space must separate individual entries on the lines. In addition to the required line types, the recording configuration file may contain blank lines and comment lines that begin with the # (number sign) character.

An **xmscheck** program, to parse and analyze a recording configuration file, is supplied as part of the Agent component. This program allows you to check the validity of a recording configuration file before it is moved to the **/etc/perf** directory. The program is described in "The xmscheck Preparser" on page 174. To check for errors, run the **xmscheck** command after creating or editing the recording configuration file. If a metric is not valid on the local system, **xmservd** or **xmtrend** will terminate processing the recording file.

### Retain Line

The retain line specifies how long time-recording files must be retained. It also defines how many days each recording file covers. The format of the retain line is as follows:

```
retain days_to_keep [days_per_file]
```

**retain**
Identifies the line.

*days_to_keep*
Must be a number greater than one. It specifies the minimum number of days a recording file must be kept before **xmservd** or **xmtrend** deletes it.

*days_per_file*
*Optional*. If specified, gives the number of days a recording file shall contain. This number must be less than or equal to **days_to_keep**. If not specified, this value defaults to the value specified for **days_to_keep**.

The ***days_to_keep*** or ***days_per_file***, or both, can also be specified by an *m* for months. Because the number of days varies from month to month, a user that wanted a recording file per month could specify this by using the *m* character.

Examples:

- To specify a recording file per week and to remove any recording file that is 12 weeks old, use the following:
```
retain 84 7
```
- To specify a recording file per month and to remove any recording file that is 1 year 1 day old, use the following:
```
retain 366 m
```
- To specify a recording file per every 3 months and to remove any recording file that is 13 months old, use the following:
```
retain m3 m13
```

Whenever **xmservd** or **xmtrend** is started and running and midnight is passed, they check to see if any of the recording files in the **/etc/perf** directory are old enough to be deleted. This is done by calculating a factor, `rf` as the integer value:

```
rf = (days_to_keep + days_per_file - 1) / days_per_file
```

If the number `d1` is the day number corresponding to the `yymmdd` part of the recording file name, and the current day number is `d2`, then the recording file is retained when the following expression is true; otherwise it is erased:

```
d2 - d1 rf x days_per_file
```

If ***days_per_file*** is greater than one, **xmservd** or **xmtrend** looks for a file with a name that indicates it is less than ***days_per_file*** old. If such a file exists, recording continues to that file. If not, a new file with a name generated from today's date is created.

When an existing recording file is opened by **xmservd** or **xmtrend**, it checks the first (configuration) record in the file. This record contains the time and date of the last modification to the recording configuration file (as of the time the recording file was created). If the recording configuration file has been modified since that time, **xmservd** or **xmtrend** begins the recording by appending a full set of control records to the file and adding the @ character to the end of the file name. Most programs that process such a file only process the part of the file up to the second set of control records.

- To rearrange the records in the file, use the **ptxmerge** program described in "The ptxmerge Merge Program" on page 96, (**xmservd** only).
- To split the file into multiple parts, use the **ptxsplit** program with the command line flag **-b**. The **ptxsplit** program is described in "The ptxsplit Split Program" on page 98 (**xmservd** only).

## Frequency Line

The frequency line sets the default sampling interval for metrics. This interval is used for all metrics that do not have a different sampling interval specified on their metric lines.

The format of the frequency line is as follows:

```
frequency interval
```

**frequency**
: Identifies the line.

*interval*
: Specifies the sampling interval in milliseconds for **xmservd** and in minutes for **xmtrend**. The value specified is rounded to the nearest multiple of the min_remote_interval value as specified with the **-i** command line argument to **xmservd** or its default value. The default recording interval for **xmtrend** is once per 10 minutes.

Recordings contain one set of statistics for each sampling interval you specify with this line type and on metric lines. It is recommended that no set of statistics ever has more than 256 metrics.

## Start-Stop Lines

The start-stop lines specify when recordings shall start and stop. Multiple lines may be used. The format of a start-stop line is as follows:

```
start dd hh mm dd hh mm
```

The first set of *dd hh mm* values specifies the time to start recording; the second set specifies the time to stop recording.

**start** Identifies the line.

*dd* Specifies the number indicating the day of the week when you want a recording to start and stop. Sunday is day number 0, Saturday is day number 6. Can be specified as a single day number, as a range such as 1-5 (Monday through Friday), or as a series of day numbers separated by commas such as 1,3,5 (Monday, Wednesday, and Friday).

*hh* Specifies the hour on a 24-hour clock (midnight is 00) when you want a recording to start and stop. Can be specified as a single hour, as a range of hours such as 07-19 (7 a.m. through 7 p.m.), or as a series of hours separated by commas such as 9,12,15 (9 a.m., 12 noon, 3 p.m.).

*mm* Specifies the minute when you want a recording to start and stop. Can be specified as a single minute value or as a series of minute values separated by commas such as 0,30 (every 30 minutes).

Exercise care when matching start and stop times -- especially when using multiple start-stop lines. It can be difficult to do this without plotting the recording intervals on a time scale. Therefore, the **xmscheck** program is available to preparse a recording configuration file and help you evaluate the resulting recording intervals.

The following examples help you understand how recording intervals are defined.

Examples:

- First, consider the following start-stop line, which causes recording to take place for 10 minutes every half hour between 9 a.m. and 6 p.m. on all weekdays. Notice that the last time recording starts every day is at 17:30 (5:30 p.m.):

  ```
  start 1-5 9-17 0,30 1-5 9-17 10,40
  ```

- If another start-stop line was added, that line would augment the first one. This is done by laying the intervals out on a time scale where all start and stop points are marked. The time scale is then processed from the beginning, creating a final set of start and stop marks by eliminating all stop marks that fall at the same minute as a start mark. Assume you supply the following two start-stop lines:

  ```
  start 1-5 9-17 0,30 1-5 9-17 10,40
  start 5 18-19 0,30 5 18-19 10,40
  ```

- This would cause recording to take place for 10 minutes every half hour between 9 a.m. and 6 p.m. on the first four weekdays and between 9 a.m. and 8 p.m. on Fridays. The same could have been specified with the following:

  ```
  start 1-4 9-17 0,30 1-4 9-17 10,40
  start 5 9-19 0,30 5 9-19 10,40
  ```

- The time scale created by **xmservd** or **xmtrend** does not wrap to the next week. Therefore, if you want recording from 11.30 p.m. to 12.30 a.m. every night of the week, you need the following two lines:

  ```
  start 0-6 23 30 1-6 00
  30 start 0 0 0 0 0 30
  ```

- For continuous recording at all times, specify the following:

  ```
  start 0 0 0 0 0 0
  ```

## Command Lines (xmservd Daemon Only)

Command lines allow the **xmservd** recording facility to execute commands or scripts when an old recording file is deleted. These commands or scripts are specified in the recording configuration file with the following format:

```
command /bin/ptxmerge /var/perf/temp %s /var/perf/year_to_date

command /bin/mv -f /var/perf/temp /var/perf/year_to_data
```

The %s in the line refers to the file to delete. The first line uses the **ptxmerge** program to merge the recording file, which is about to be deleted with the **year_to_date** file of accumulated recording files, and place the output from the merge to a temporary file. The last line moves the temporary file over to the previous **year_to_date** file. Note that this series of commands is not safe; it is meant only to illustrate the facility. To perform the previous task, use a script that ensures there is adequate disk space so that you do not lose data.

## Metric Lines

One metric line must be supplied for each metric you want recorded or a wildcard can be used to specify a group of metrics. For more information on wildcards, see the chapter on Using Wildcards in the Configuration File. The metric lines have the following format:

metric_name [*interval*]

**metric_name**
Must be the full path name of a statistic. Because **xmservd** or **xmtrend** can only access local statistics, the path name must not include the hosts part of the path name. The path name does not begin with a / (slash).

Process contexts have a name consisting of the process ID, a ~ (tilde), and the name of the executing program. To reach a statistic for a specific process, you can specify the process context name as either the process ID followed by the tilde, or the name of the executing program. The following example shows how to specify a statistic for the wait pseudo process, which, on AIX Version 3.2, always has a process ID of 514. Both lines point to the same statistic.

```
Proc/514~/usercpu
Proc/wait/usercpu
```

If you specify a name of a program currently executing in more than one process, only the first one encountered is used. Generally, recording of process statistics from **xmservd** or **xmtrend** is discouraged except for processes that are expected to never die. If a process dies, it is deleted from the statset and is not added back, should the process be restarted later.

*interval*
*Optional*. If specified, defines the sampling interval in milliseconds to use for recording this metric. If omitted, the metric is recorded with the sampling interval specified on the frequency line.

Use the **xmpeek** command to determine which local metrics can be recorded.

## Using Wildcards in the Configuration File (xmtrend and xmservd Recording)

With wildcards, users can specify groups of metrics without having to specify each metric individually. In the following example, individual processes without using wildcards would be listed as:

```
Proc/2156~X/cpupct
Proc/3216~X/cpupct
```

To use wildcards in **xmtrend** and **xmservd** the following specifies *all* processes:

```
Proc/*/cpupct
```

Or, the following form specifies all process names starting with ora:

```
Proc/ora*/cpupct
```

The number of processes is limited by the total number of statistics limit within **xmtrend**, which is 256. When **xmtrend** reads and resolves the configuration file, no new processes are evaluated and added to the trend list.

Similarly, using wildcards can be applied to any parent context with multiple instances, such as the following example:

```
CPU/[cpuid], PagSp/[disk], Disk[diskid], LAN/[adapterid], [volume group]/[diskid],
  [volume group]/[lvid],NetIF/[adapterid]
```

If a user wants to record the `kern` and `user` statistics for all the processors, the metric line in the configuration file is displayed as follows:

```
CPU/*/kern  CPU/*/user
```

In a two-CPU system, this expands to the following:

```
CPU/cpu0/kern
CPU/cpu1/kern
CPU/cpu0/user
CPU/cpu1/user
```

To record the data `busy` for all the disks on your system, the metric line would be as follows:

```
Disk/*/busy
```

Depending upon the number of active hdisks in the system, this expands to the following:

```
Disk/hdisk0/busy
Disk/hdisk1/busy
Disk/hdisk2/busy
```

## Hot Lines (xmservd Recording Only)

HotSets allow metrics to be monitored by activity rather than by name. HotSets are defined by the following format. The values correspond to the arguments of the SpmiAddSetHot subroutine call:

```
# format of line to define hotfeed followed by examples:

#key                     max   thres  freq                             seve  trap
#word metric             resp  hold   uency  feed_type  except_type  rity  no

hot  LAN/*/framesin       1       0    60000  Always
hot  Disk/*/busy          3      50    10000  Threshold  Trap           0     14
hot  FS/*/%totfree        3      95   300000  Threshold  Both           4     16
hot  FS/rootvg/*/%totfree 0      95   300000  Always
hot  RTime/LAN/*/above99  3      80   300000  Threshold  Exception      2
```

**hot** The keyword indicating HotSet recording.

**metric** The metric with a wildcard in the specification.

**maxresp** The maximum number of responses to record. If the feed_type is Threshold, this value must be greater than one. One exception/trap is sent for each metric that exceeds the threshold up to the maximum value of this field.

**threshold** If feed_type is set as the Threshold, this field is the value that must be exceeded for the exception/trap to be sent. If the value is specified as a negative number, the threshold is considered to be exceeded if the monitored metric is less than the numeric threshold value.

**frequency** The frequency to monitor the metrics. This value is in milliseconds.

**feed_type** The valid types are: Always or Threshold.

**exception_type**
The valid types are: Exception, Trap, or Both.

**severity** If sending an exception, the severity level for the exception.

**trap_number** If sending a trap, the trap number to send.

Each line defines a separate HotSet. No more than MAX_HOT_COUNT (40) Hot events will be processed at any given time. Use HotSets to monitor thresholds that represent unusual performance behavior, and not the usual.

Use the **ptxhottab** and **ptx2stat** recording support programs to process **xmservd** recording files that contain HotSet values.

The following is an example of an **xmservd** recording configuration file:

```
# SAMPLE RECORDING CONFIGURATION FILE
# Keep files at least 7 days and let each file contain
# two day's recordings
retain 7 2
# Set default sampling interval to one minute
frequency 60000
# Give five statistics to record with default frequency
CPU/cpu0/user
CPU/cpu0/kern
Mem/Real/sysrepag
Mem/Virt/pagein
Mem/Virt/steal
# Two additional statistics are recorded every 20 seconds
IP/NetIF/tr0/ioctet 20000
IP/NetIF/tr0/ooctet 20000
# record every weekday from 8.30 a.m. to 5 p.m., except during
# the lunch hour from noon to 1 p.m.
start 1-5 8 30 1-5 17 0
start 1-5 13 0 1-5 12 0
```

## Selecting Metrics for the Recording Configuration File

The **xmpeek** program is supplied as part of the Performance Toolbox for AIX. The command line is as follows:

```
xmpeek -l
```

If the **xmpeek** program is invoked with the **-l** flag (lowercase L), it lists all the available statistics of the local host. The list of statistics is sent to standard output, which permits you to redirect it to a file or pipe it into another command. The following example shows a partial listing of statistics:

```
/hostname/CPU/              Central processor statistics
/hostname/CPU/gluser        System-wide time executing in user mode (percent)
/hostname/CPU/glkern        System-wide time executing in kernel mode (percent)
/hostname/CPU/glwait        System-wide time waiting for IO (percent)
/hostname/CPU/glidle        System-wide time CPU is idle (percent)
/hostname/CPU/glnice        System-wide time CPU is running w/nice priority (%)
  . . .
```

When a host's statistics include contexts that may exist in multiple instantiations and such instantiations are volatile, the list does not break all such contexts down in their components. Rather, only the first instance of the context is broken down and all further instances are listed with five dots appended to the statistics path name, as shown in the following example. The process identified by `514~wait` (actually a pseudo process) is fully broken down. All other processes are listed only with their identifiers because they would all break down to the same base statistics as the wait process.

```
/hostname/Proc/              Process statistics
/hostname/Proc/pswitch       Process context switches
/hostname/Proc/runque        Average count of processes waiting for the CPU
/hostname/Proc/runocc        Number of samplings of runque
/hostname/Proc/swpque        Average count of processes waiting to be paged in
/hostname/Proc/swpocc        Number of samplings of swpque
/hostname/Proc/ksched        Number of kernel process creations
```

```
/hostname/Proc/kexit              Number of kernel process exits
/hostname/Proc/514~wait/          Process wait  (514) %cpu 54.6, PgSp: 0.0mb, uid:
/hostname/Proc/514~wait/pri       Process priority
/hostname/Proc/514~wait/wtype     Process wait status
/hostname/Proc/514~wait/majflt    Process page faults involving IO
/hostname/Proc/514~wait/minflt    Process page faults not involving IO
/hostname/Proc/514~wait/cpums     CPU time in milliseconds in interval
/hostname/Proc/514~wait/cpuacc    CPU time in milliseconds in life of process
/hostname/Proc/514~wait/cpupct    CPU time in percent in interval
/hostname/Proc/514~wait/usercpu   Process CPU use in user mode (percent)
/hostname/Proc/514~wait/kerncpu   Process CPU use in kernel mode (percent)
/hostname/Proc/514~wait/workmem   Physical memory used by process private data (4K)
/hostname/Proc/514~wait/codemem   Physical memory used by process code (4K pages)
/hostname/Proc/514~wait/pagsp     Page space used by process private data (4K page
/hostname/Proc/514~wait/nsignals  Signals received by process
/hostname/Proc/514~wait/nvcsw     Voluntary context switches by process
/hostname/Proc/514~wait/tsize     Code size (bytes)
/hostname/Proc/514~wait/maxrss    Maximum code+data resident set size (4K pages)
/hostname/Proc/12002~x/.....
/hostname/Proc/13207~xlock/.....
/hostname/Proc/771~netw/.....
/hostname/Proc/1~init/.....
/hostname/Proc/5723~trapgend/.....
/hostname/Proc/0~/.....
/hostname/Proc/15339~aixterm/.....
/hostname/Proc/2823~syncd/.....
/hostname/Proc/13047~xmtrend/.....
/hostname/Proc/15593~aixterm/.....
 . . .
```

Strip the hostname from the description on the right of the statistic when using statistics in configuration file. For example, the following information:

```
/hostname/Mem/Virt/pageout         4K pages written by VMM
/hostname/Proc/514~wait/cpupct     CPU time in percent in interval
```

would be placed in the **xmtrend** configuration file as follows:

```
Mem/Virt/pageout Proc/514~wait/cpupct
```

## The xmscheck Preparser

When **xmservd** is started with the **-v** command line argument, its recording configuration file parser writes the result of the parsing to the log file. The output includes a copy of all lines in the recording configuration file, any error messages, and a map of the time scale, indicating when recording starts and stops.

Although the log file is useful to document what is read from the recording configuration file, it is not a useful tool for debugging of a new or modified recording configuration file. Therefore, the **xmscheck** program is available to preparse a recording configuration file before you move it to the **/etc/perf** directory, where **xmservd** and **xmtrend** look for the recording configuration files.

When **xmscheck** is started without any command line argument, it parses the **/etc/perf/xmservd.cf** file. You can therefore determine how the running daemon is configured for recording. For **xmtrend** recording files, specify on the command line the file to parse.

Output from **xmscheck** goes to standard output. The parsing is done by the same module that does the parsing in **xmservd** and **xmtrend**. That module is linked in as part of each program. The parsing checks that all statistics specified are valid and prints the time scale for starting and stopping recording in the form of a time table.

In the time table, each minute has a numeric code as follows:

**0** Recording is inactive. Neither a start nor a stop request was given for the minute.

**1** Recording is active. Neither a start nor a stop request was given for the minute.

**2** Recording is inactive. A stop request was given for the minute.

**3** Recording is active. A start request was given for the minute.

The following Sample **xmscheck** Time Table shows how **xmscheck** formats the time table. Only the part of the table that covers Tuesday is shown.

```
Day 2, Hour 00:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 01:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 02:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 03:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 04:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 05:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 06:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 07:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 08:
000000000000000000000000000000311111111111111111111111111111
Day 2, Hour 09:
111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 10:
111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 11:
111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 12:
200000000000000000000000000000000000000000000000000000000000
Day 2, Hour 13:
311111111111111111111111111111111111111111111111111111111111
Day 2, Hour 14:
111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 15:
111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 16:
111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 17:
200000000000000000000000000000000000000000000000000000000000
Day 2, Hour 18:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 19:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 20:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 21:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 22:
000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 23:
000000000000000000000000000000000000000000000000000000000000
```

## Starting Recording Sessions from the xmtrend Command Line

The **xmservd** daemon is always started from the **inetd** daemon. For a full description on how to start and stop **xmservd** as well as the command line options, see Chapter 13, “Monitoring Remote Systems,” on page 153. The **xmtrend** command line options are described here, but because **xmtrend** uses the SPMI library and shared memory, refer to “Releasing Shared Memory Manually” on page 155.

The **xmtrend** agent can be started from the command line or near the end of the **/etc/inittab** file. The general format of the command line is as follows:

```
xmtrend {-f infile} {-d recording_dir}
        {-n recording_name} {-t trace_level}  {-T}
```

The following flags can be specified when starting **xmtrend**. All command line options are optional.

**-f**
: Allows the user to specify a configuration file to use, instead of the default. By default if the **-f** is not used, **xmtrend** looks for and uses **/etc/perf/xmtrend.cf** as the configuration file. A configuration file must be available so **xmtrend** knows what to monitor.

**-d**
: Specifies the output directory for the recording files. The default is to place the recording files in the **/etc/perf** directory.

**-n**
: Specifies a name for the recording file. By default, **xmtrend** creates recording files named **xmtrend.***some date*. If `-n myrecording` is specified, the recording files will be named **myrecording.***some date*

**-t**
: Specifies a trace level. **xmtrend** prints various information to a log file in **/etc/perf**. The trace level can be set from 1 to 9. The higher the trace level, the more trace data is generated. This trace data is useful to determine **xmtrend** recording status and for debugging purposes. The log file name is either **xmtrend.log1** or **xmtrend.log2**. **xmtrend** will cycle between these two files after a file reaches the maximum size.

**-T**
: Enables top processing to support the **jtopas** client. This option must be used for top recordings and near real-time data support. The **xmtrend** daemon uses the shipped **jtopas.cf** configuration file for all recordings and will place them in the **/etc/perf/Top** directory. The configuration file's metrics-list section for all recordings is fixed and cannot be changed.

## Session Recovery by the xmtrend Agent

If the **xmtrend** agent is terminated then restarted, **xmtrend** examines the recording files in **/etc/perf** or the directory specified by the **-d** flag. If a recording file exists with the current date, **xmtrend** appends to this file and continues to write to the recording file. Otherwise it creates a new recording file.

# Chapter 15. SNMP Multiplex Interface

The SNMP (Simple Network Management Protocol) is a network protocol based upon the Internet protocol. As its name implies, its main purpose is to provide a protocol that allows management of networks of computers. Programs based upon SNMP are currently dominating the network management arena in non-SNA environments. One set of commonly used SNMP-based network management programs are the programs in NetView.

## Network Management Principles

Network management is primarily concerned with the availability of resources in a network. As implemented on top of SNMP, it uses a client/server model where one or a few hosts in the network run the client programs (known as SNMP Managers) and all network nodes (if possible) run the server code. On most host types the server code is implemented as a daemon, **snmpd**, usually referred to as the SNMP Agent.

Communication between the SNMP manager and the SNMP daemon uses two protocol models. The first model is entirely a request/response type protocol; the other is based upon traps, which are unsolicited packets sent from a server (agent) to the client (manager) to inform of some event.

The request/response protocol supports three request types:

**Get**
Issued from the manager to an agent, requesting the current value of a particular variable. The agent will return the value if it is available.

**Set**
Issued from the manager to an agent, requesting the change of a particular variable. By implication, the changing of a value will be interpreted by the agent as also meaning that the change of the value must be enforced. For example, if the number of memory buffers is changed, the agent is expected to implement this change on the system it runs on. A large number of system variables cannot be set but are read-only variables.

**Get next**
Issued from the manager to the agent, requesting the agent to go one step further in the hierarchy of variables and return the value of the next variable.

As is implied by the "get next" request type, variables are arranged in a hierarchy much like the hierarchy used to maintain the statistics provided by the System Performance Measurement Interface (SPMI) and the **xmservd** daemon. Unlike the SPMI context hierarchy, however, even though an SNMP manager can traverse the hierarchy of variables to see what's available, it identifies those variables by a decimal coding system and is not able to convert these codes to textual descriptions by itself. To make the SNMP manager able to translate decimal coding into text, you must provide a file that describes the variables and the hierarchy. The file must describe the variables in a subset of the Abstract Syntax Notation (ASN.1) as defined by ISO. The subset used by SNMP is defined in RFC 1065. A file that describes a set or subset of variables and the hierarchy is referred to as a MIB file because it is said to describe a management information base (MIB).

Usually, an SNMP agent will know what variables it is supposed to provide and uses a fixed set. In other situations, the SNMP agent's set of variables may need to be expanded because special programs or special hardware is installed. This can be done through a programming interface called SMUX (SNMP Multiplex). The remainder of this topic describes how SMUX is used by the **xmservd** daemon to expand the set of variables available from the SNMP agent.

**Note:** The interface between **xmservd** and SMUX is only available on IBM RS/6000 Agents.

© Copyright IBM Corp. 1994, 2004

## Interaction Between xmservd and SNMP

The objective of the **xmperf** program suite is much different from that of the NetView programs. The latter are concerned primarily with supervision and corrective action aiming at keeping the network resources available and accessible. Generally, resource availability is of more concern than resource utilization.

The **xmperf** program suite is primarily concerned with the continuous monitoring of resource utilization, aiming at:

- Identifying and possibly improving performance-heavy applications.
- Identifying scarce system resources and taking steps to provide more of those resources.
- Predicting loads as input to capacity planning for the future.
- Identifying acute performance culprits and taking steps to resolve the problems they cause.

Somewhere between the two products is a vaguely defined area in which both are interested. This means that certain of the variables (or statistics) must be available in both environments. It also means that if the two products do not share information, they both access the same information, inducing an overhead that could be eliminated if they had a common access mechanism.

Such a common access mechanism is available through the xmservd/SMUX interface. It allows the **xmservd** daemon to present all its statistics to the SNMP agent as read-only variables. The **xmservd/SMUX** interface is invoked by placing a single stanza in the configuration file**/etc/perf/xmservd.res**. See “Files used by xmservd” on page 272 for alternative locations for **xmservd.res**. The stanza must begin in column one of a line of its own and must be:

```
dosmux
```

When the **dosmux** stanza is in effect, every statistic available to the **xmservd** daemon is automatically registered with the **snmpd** daemon on the local host. Dynamic data suppliers can add to or delete from the hierarchy of statistics. Any changes induced by dynamic data suppliers are communicated to the **snmpd** daemon. Every 15 seconds **xmservd** checks for such changes.

The **xmservd** daemon can produce an MIB file that describes all the variables currently exported to **snmpd**. This is done whenever you send a **SIGINT** (kill -2) to the **xmservd** process. The MIB file is created in ASN.1 notation and placed in **/etc/perf/xmservd.mib**. Any old copy of the file is overwritten. The generated MIB file can be moved to the host where NetView runs and imported by NetView.

When you need to generate a MIB file by sending a **SIGINT** to the **xmservd** daemon, make sure you have all relevant dynamic data-supplier programs running and registered with the daemon. Also have at least one data consumer registered with the daemon. This makes sure the generated MIB file includes all possible statistics in your host.

To be able to test the **xmservd/SMUX** interface with the SNMP program **snmpinfo** it is convenient to have a makefile for easy updating of the MIB file used by **snmpinfo**. This file is **/etc/mib.defs**. You must be root to update it. Consequently, you must have root authority to run the makefile.

A sample makefile to update the **snmpinfo** MIB file is shown in the following example and is available in **/usr/samples/perfagent/server/Make.mib**:

```
all: /etc/mib.defs.org /etc/mib.defs
/etc/mib.defs.org:
  cp /etc/mib.defs /etc/mib.defs.org
/etc/mib.defs: /etc/perf/xmservd.mib
  mosy -o /tmp/mib.defs /etc/perf/xmservd.mib
  cp /etc/mib.defs.org /etc/mib.defs
  cat /tmp/mib.defs >> /etc/mib.defs
  rm /tmp/mib.defs
```

## SMUX Configuration Conflicts

During the installation, and provided the **snmp** option of the base system's **bosnet** or **tcpip** component is installed, two configuration files are updated to allow the **xmservd** daemon to connect to the SNMP agent. The two files are updated with default identification strings and passwords.

```
/etc/snmpd.conf:
smux 1.3.6.1.4.1.2.3.1.2.1.3 xmservd_pw # xmservd
/etc/snmpd.peers:
"xmservd" 1.3.6.1.4.1.2.3.1.2.1.3 "xmservd_pw"
```

If the **xmservd/SMUX** interface does not work as intended, even after you inserted **dosmux** in the file **/etc/perf/xmservd.res**, there may be a conflict between the **xmservd/SMUX** strings and other strings in the configuration files. To resolve the problem, simply change the last digit in the identification string for **xmservd** to something unique, then restart the **snmpd** daemon.

# Limitations Induced by SMUX

One of the advanced features of the SPMI context hierarchy is that it allows you to instantiate in multiple levels. One context may define disks and the actual number of disks varies from host to host. Through instantiation, subcontexts are added for each disk present in a particular host.

The SNMP data structures allow for a similar facility, namely the definition of tables. In the previous case, the table would be “Disks” and it would contain as many elements as there were disk drives, each element containing all the fields defined for a disk.

With the SPMI interface, you can continue the instantiation at the next level in the context hierarchy. For example, each disk may have a variable number of logical volumes assigned to them, each with its identical set of statistics. Instantiation would then allow you to adjust the context hierarchy as logical volume assignment changes.

SNMP does not allow such a thing. A table is the only type of structure that can be instantiated, and it must always be at the lowest level in the hierarchy. Because of this, the SPMI context hierarchy has been adjusted so it only in one case instantiates in multiple levels. Otherwise, it would not be possible to export the context hierarchy to the SNMP agent.

The one situation where the SPMI instantiates in multiple levels is the context **FS**. When this context is exported to **snmpd**, only the top-level is exported. Statistics for individual file systems are not available through **snmpd**.

## SMUX Instantiation

Because of the differences between SPMI and the MIB definitions when it comes to instantiation, it seems warranted to illustrate what instantiation looks like in the two cases. This is illustrated by looking at the instantiation of disk drives.

The following example shows the list of disk statistics clipped from the output of the command **xmpeek -l**. Notice that each disk (there are three of them) has four statistics defined:

```
/nchris/Disk/                   Disk and CD ROM statistics
/nchris/Disk/hdisk0/            Statistics for disk hdisk0
/nchris/Disk/hdisk0/busy        Time disk is busy (percent)
/nchris/Disk/hdisk0/xfer        Transfers to/from disk
/nchris/Disk/hdisk0/rblk        512 byte blocks read from disk
/nchris/Disk/hdisk0/wblk        512 byte blocks written to disk
/nchris/Disk/hdisk1/            Statistics for disk hdisk1
/nchris/Disk/hdisk1/busy        Time disk is busy (percent)
/nchris/Disk/hdisk1/xfer        Transfers to/from disk
/nchris/Disk/hdisk1/rblk        512 byte blocks read from disk
/nchris/Disk/hdisk1/wblk        512 byte blocks written to disk
/nchris/Disk/hdisk2/            Statistics for disk hdisk2
/nchris/Disk/hdisk2/busy        Time disk is busy (percent)
```

```
/nchris/Disk/hdisk2/xfer           Transfers to/from disk
/nchris/Disk/hdisk2/rblk           512 byte blocks read from disk
/nchris/Disk/hdisk2/wblk           512 byte blocks written to disk
```

The SNMP perception of this context structure is somewhat different. As the structure is exported from **xmservd** through the SMUX interface it is converted to an MIB table. This structure is illustrated in Figure 14 on page 227.

Type the following command:

```
snmpinfo -md -v xmdDisk
```

To print output as shown in the following example:

```
xmdDiskIndex.0 = 1
xmdDiskIndex.1 = 2
xmdDiskIndex.2 = 3
xmdDiskInstName.0 = "hdisk0"
xmdDiskInstName.1 = "hdisk1"
xmdDiskInstName.2 = "hdisk2"
xmdDiskBusy.0 = 20943
xmdDiskBusy.1 = 679
xmdDiskBusy.2 = 386
xmdDiskXfer.0 = 11832
xmdDiskXfer.1 = 444
xmdDiskXfer.2 = 89
xmdDiskRblk.0 = 73201
xmdDiskRblk.1 = 2967
xmdDiskRblk.2 = 6595
xmdDiskWblk.0 = 137449
xmdDiskWblk.1 = 1585
xmdDiskWblk.2 = 105
```

As you can see, the retrieval sequence is inverted. Where the SPMI retrieves all statistics for one disk before proceeding to the next disk, SMUX traverses the structure by reading one statistic for all disks before proceeding to the next statistic.

You'll see that for each disk instance, an artificial statistic is created to provide the index of each value (with the name **xmdDiskIndex**).

Also notice how the name of the instance (in this case the name of the disk drive) is displayed as another artificial type of statistic, which always has the name **InstName** meaning "instance name."

The MIB definition for disk statistics is shown in the "Example MIB Description for Disk Instantiation" on page 181.

## Instantiation Rules

In SPMI, a context can be defined as having an instantiation type of:

**SiNoInst**
Context is never instantiated, not even if requested.

**SiCfgInst**
Context is instantiated when **xmservd** is started. Further attempts to instantiate are done only when explicitly requested. Most data-consumer programs will not attempt to instantiate contexts with this context type; **xmperf** does not. Examples of contexts with this instantiation type are disks and page spaces.

**SiContInst**
Context is instantiated when it is created and when instantiation is requested. Most data-consumer programs should attempt to instantiate contexts with this context type; **xmperf** does. The classical example of a context with this instantiation type is the context defining processes.

When exporting contexts through SMUX, contexts with instantiation type of **SiCfgInst** or **SiContInst** are converted to tables.

For dynamic data-supplier programs, a special restriction applies to the use of **SiCfgInst** and **SiContInst**. Neither can be used for contexts that are at the top of the hierarchy of non-volatile contexts defined by a dynamic data supplier (DDS). Also, neither may be used for contexts that are added as volatile extensions.

Generally, because a request for instantiation is not passed to a dynamic data-supplier program, avoid using anything but **SiNoInst** in your DDS programs. If you want to use **SiContInst**, all of the subcontexts of the context with **SiContInst** should be volatile contexts of the same type.

## Example MIB Description for Disk Instantiation

```
xmdDisk  OBJECT-TYPE
         SYNTAX  SEQUENCE  OF  XmdDisk
         ACCESS  not-accessible
         STATUS  mandatory
         DESCRIPTION
             "Disk  and  CD  ROM  statistics
         ::=  {xmd  4  }
xmdDiskEntry  OBJECT-TYPE
         SYNTAX XmdDiskEntry
         ACCESS  not-accessible
         STATUS  mandatory
         DESCRIPTION
             "Element  of  above  table"
         ::=  {xmdDisk  1  }
XmdDiskEntry  ::=
         SEQUENCE
         {
             xmdDiskIndex  INTEGER,
             xmdDiskInstName  DisplayString,
             xmdDiskBusy  Counter,
             xmdDiskXfer  Counter,
             xmdDiskRblk  Counter,
             xmdDiskWblk  Counter
         }
xmdDiskIndex  OBJECT-TYPE
         SYNTAX  INTEGER
         ACCESS  read-only
         STATUS  mandatory
         DESCRIPTION
             "Index  Number"
         ::=  {xmdDiskEntry  1  }
xmdDiskInstName  OBJECT-TYPE
         SYNTAX  DisplayString
         ACCESS  read-only
         STATUS  mandatory
         DESCRIPTION
             "Instance  Name"
         ::=  {xmdDiskEntry  2  }
xmdDiskBusy  OBJECT-TYPE
         SYNTAX  Counter
         ACCESS  read-only
         STATUS  mandatory
         DESCRIPTION
             "Time  disk  is  busy  (percent)"
         ::=  {xmdDiskEntry  3  }
     .
     .
     .
xmdDiskWblk  OBJECT-TYPE
         SYNTAX  Counter
         ACCESS  read-only
         STATUS  mandatory
         DESCRIPTION
             "512  byte  blocks  written  to  disk"
         ::=  {xmdDiskEntry  6  }
```

# Chapter 16. Data Reduction and Alarms with filtd

The **filtd** program is designed to run as a daemon. It takes three command line arguments, all of which are optional:

**filtd** [**-f** *config_file*] [**-b** *buffer_size*] [**-p** *trace_level*]

Command Line Arguments for filtd:

**-f**
Overrides the default configuration file name. If this option is not given, the file name is assumed to be available in **/etc/perf/filter.cf** or else as described in Appendix B, "Performance Toolbox for AIX Files," on page 271. The configuration file is where you tell **filtd** what data reduction and alarm definitions you want.

**-p**
Specifies the level of detail written to the log file. The trace level must be between 1 and 9. The higher the trace level the more is written to the log file. If this option is not specified, the trace level is set to zero.

**-b**
Buffer size for communications with **xmservd** via RSI. The default buffer of 2048 bytes will allow for up to 60 statistics to be used in defining new statistics and alarms. If more are needed, the buffer size must be increased. It may also be necessary to increase the **xmservd** buffer size.

## filtd Configuration File

When **filtd** is started, it immediately issues an **RSiOpen()** call (see the **RSiOpen** subroutine) to register with the local **xmservd** daemon. This causes **xmservd** to start if it is not already running. Following a successful connection to **xmservd**, **filtd** then reads the configuration file and parses the information you supplied in the file.

The configuration file contains expressions, which either define new statistics from existing ones or define alarms from statistics. Each time the name of a statistic is encountered while parsing an expression, it is checked with the **xmservd** daemon whether it is valid. If not, the entire expression is discarded and **filtd** proceeds to parsing the next expression in the configuration file, if any. Errors detected are reported to the log file.

When all expressions have been parsed, **filtd** processes all expressions that define new statistics. First it registers the subscription for statistics it needs to build the new ones with **xmservd**. Then it registers with **xmservd** as a dynamic data supplier. At this point, **filtd** is both a consumer and a supplier of statistics. At the end of this initialization phase, **filtd** instructs **xmservd** to start feeding the statistics it subscribed to.

The next phase runs through any alarm definitions. No new statistics are defined at this point, but because this is the last of the initialization phases, alarms may refer to statistics that are defined by the previous phase.

### Sampling Interval

Whenever new statistics are defined through the **filtd** configuration file, raw data statistics are initially requested from xmservd every five seconds. As long as no data-consumer program subscribes to the new statistics, the sampling interval remains at five seconds or some smaller value as required to meet the minimum requirements for alarm duration as described in "Alarm Duration and Frequency" on page 188.

When other data-consumer programs subscribe to one or more of the new statistics, the sampling interval is adjusted to match the data-consumer program that requires the fastest sampling. Again, if the requirements of an alarm's duration dictates a smaller interval, that is selected.

For most purposes, sampling intervals can safely be set at two seconds or more. Be aware that if you have defined thirty new statistics but subscribe to only one, all thirty are calculated each time you are sent a data feed for the one you subscribe to.

© Copyright IBM Corp. 1994, 2004

## Automatic Start of filtd

Since **filtd** is a dynamic data-supplier program, you may want to always have it running when the **xmservd** daemon runs. You can cause this to happen if you add a line to the **xmservd** configuration file, specifying the full path name of the **filtd** program and any command line arguments. For example:

```
supplier: /usr/bin/filtd -p5
```

## Termination of filtd

The **filtd** daemon can be terminated by killing its process (but don't use **kill -9**). The daemon will terminate itself if it has not received data_feed packets from **xmservd** for 10 times the data feed interval. This ensures that **filtd** is terminated whenever **xmservd** is.

## Data Reduction

Although the term *data reduction* is used, you can actually use the data reduction facilities of filtd to do exactly the opposite. You can define as many new statistics as you want to. However, the most common use of the data reduction facility will likely be to reduce a large number of statistics to a reasonable set of combined values.

Whether you define lots of new statistics or combine existing ones into fewer new ones, you do it by entering expressions into the configuration file. The general syntax format of expressions for defining new statistics is `target = expression description` where **target** is the unqualified name of non-existing variable.

The expression must start with an alpha and contain only alpha-numeric characters and percent signs and is in the form {*variable|wildcard|const*} **operator** {*variable|wildcard|const*}.

A **variable** must be a fully qualified **xmperf** variable name with slashes replaced by underscores. Valid names have at least one underscore. The first name component must start with an alpha character and subsequent names may also begin with a percent sign. All must contain only alpha-numeric characters, a percent sign, a tilde (~), a period, an underscore preceded by an escape character (the backslash '/'), or a wildcard. The referenced variable must already exist (cannot be defined in this configuration file).

A **wildcard** is a fully qualified **xmperf** variable name with slashes replaced by underscores. Valid names have at least one underscore. The first name component must start with an alpha character, and subsequent names may also begin with a percent sign. All must contain only alpha-numeric characters and percent sign or must be a wildcard The wildcard character must appear in place of a context name, must only appear once, and must be one of the characters '+', '*', '#', '>', '<'.

An **operator** is one of *, **/**, **%**, or **+**.

The **const** consists of [*digits*].

The **digits description** is text describing the defined target variable. The descriptio must be enclosed in double quotation marks and the length of the text should not exceed 64 characters.

The expression can contain as many parentheses as are required to make the expression unambiguous. It is a good idea to use parentheses liberally if you are in doubt. If you are uncertain how your expression is interpreted, run the program with the command line option **-p5**. This writes the interpretation of the expression to the log file. If the interpretation is not what you intended, add parentheses.

All numeric constants you specify in an expression are evaluated as floating-point numbers. Similarly, the resulting new statistics (the "target" statistics) are always defined as floating-point numbers.

All new statistics are added to the context called **DDS/IBM/Filters** so that a new statistic called "avgload" would be known to data-consumer programs as **DDS/IBM/Filters/avgload**.

## Wildcards

The use of *wildcards* is a way of referring to multiple instances of a given statistic with one name but, more important, it makes your expression independent of the actual configuration of the system it is used on. For example, the expression:

```
allreads= Disk_+_rblk
```

could evaluate to different expressions on different machines, such as:

```
allreads =((Disk/cd0/rblk + Disk/hdisk1/rblk) + Disk/hdisk0/rblk)
allreads = Disk/hdisk0/rblk
```

The possible wildcard characters and their meaning are as follows:

| Character | Meaning |
|---|---|
| + | All values matching the wildcard are added together. |
| * | All values matching the wildcard are multiplied with each other. Note that unless all the values are non-zero, the result will be zero. |
| # | Evaluates to a constant, which is the number of values that match the wildcard. |
| > | Evaluates to the maximum value of all those matching the wildcard. |
| | Evaluates to the minimum value of all those matching the wildcard. |

## Quantities and Counters

As described in the discussion of how to define statistics in System Performance Measurement Interface API (Chapter 18, “System Performance Measurement Interface Programming Guide,” on page 201) a statistic provided by the SPMI is either of type **SiCounter** or of type **SiQuantity**. You can combine the two types in expressions to define new statistics, but the resulting statistics as added by **filtd** are always defined as of type **SiQuantity**.

This has consequences you need to understand in order to define and interpret new statistics. To see how it works, assume you have a raw statistics value defined as a counter. If data feeds for a raw statistic from **xmservd** called *widgets* are received with an interval of two seconds, you might get the results illustrated in the following table:

| Elapsed seconds | Counter value | Delta value | Calculated rate/second |
|---|---|---|---|
| 0 | 33,206 | | |
| 2 | 33,246 | 40 | 20 |
| 4 | 33,296 | 50 | 25 |
| 6 | 33,460 | 164 | 82 |
| 8 | 33,468 | 8 | 4 |
| 10 | 33,568 | 100 | 50 |

If you define a new statistic with the expression:

```
gadgets = widgets
```

and use **xmperf** to monitor this new statistic, you will always see the rate as it was calculated when the latest data feed was received. The following table shows what you see with different viewing intervals:

| Elapsed seconds | Interval 1 second | Interval 2 seconds | Interval 4 seconds | Raw rate at 4 seconds |
|---|---|---|---|---|
| 1 | ? | | | |
| 2 | 20 | 20 | | |
| 3 | 20 | | | |
| 4 | 25 | 25 | 25 | 23 |
| 5 | 25 | | | |
| 6 | 82 | 82 | | |

```
  7       82
  8        4         4           4        43
  9        4
 10       50        50
```

The last column in the previous table shows what the values would have been at four-second intervals if the raw counter value had been used to arrive at the average rate. Obviously, you need to take this into consideration when you define new statistics. The best way is to standardize the intervals you use.

To summarize, when new values are defined by you, any raw values of type **SiQuantity** are used as they are while the latest calculated rate per second is used for raw values of type **SiCounter**.

## Data Reduction Delay

Because **filtd** must read the raw statistics before it can calculate the values of the new ones, the new statistics are always one "cycle" behind the raw statistics. An **xmperf** instrument that plots a statistic you defined along with the raw statistics used to calculate it always shows a time lag between the new value and the raw ones. This is obvious when the **filtd** program receives data feeds at the same speed as the **xmperf** instrument does, however, whether you see it or not, the delay is always effective.

If you want to see what it looks like, put only the following line in the **filtd** configuration file:

```
user = CPU_cpu0_user
```

and then define an instrument in **xmperf** to display the values:

```
CPU/cpu0/user
DDS/IBM/Filters/user
```

## Data Reduction Examples

The **xmservd** daemon divides usage of the CPU resource on IBM RS/6000 systems into four groups: kernel, user, wait, and idle. If you wanted to present it as only two: busy and notbusy, you could define those two new statistics with the following expressions.

```
busy = CPU_cpu0_kern + CPU_cpu0_user "CPU running"
notbusy = CPU_cpu0_wait + CPU_cpu0_idle "CPU not running"
```

If you want to see the average number of bytes per transmitted packet for an IP interface, your expression would be:

```
packsize = IP/NetIf_tr0_ooctet / IP/NetIf_tr0_opacket \
           "Average packet size"
```

In the previous example, the divisor may often be zero. Whenever a division by zero is attempted, the resulting value is set to zero. The example also shows that expressions can be continued over more than one line by terminating each line except the last one with a \ (backslash).

If you want to see how large a percentage of the network packets are using the loopback interface in your system, try a definition like the following:

```
localpct = (IP/NetIf_lo0_ipacket + IP/NetIf_lo0_opacket) * 100 \
           / (IP/NetIf_+_ipacket + IP/NetIf_+_opacket)   \
           "Percent of network packets on loopback interface"
```

The previous example iillustrates the usefulness of wildcards. Another, more advanced use of wildcards is shown in the following example. The new value is **readdistr** and will hold the average percent of reads from all disks expressed as a percentage of the reads from the disk that had the most reads.

```
readdistr = (Disk_+_rblk / Disk_#_rblk) * 100 / (Disk_>_rblk) \
     "Average disk reads in percent of most busy disk"
```

## Rounding

All calculations are done in floating-point. Rounding occurs when a data-consumer program defines the receiving field as **SiLong**. Most data-consumer programs use the standard function **RSiGetValue()** to retrieve the fields. This function rounds the data values when they are retrieved. If you display raw values that are supplied in floating-point and values computed from these values, then you may get rounded values, which seem to be wrong.

For example, two raw values may be 4.3 and 2.4, which would usually be displayed as 4 and 2, but the product computed by **filtd** would be 4.3 x 2.4 = 10.32 (rounded to 10 when displayed) rather than 4 x 2 = 8.

## Defining Alarms

An alarm consists of an *action* part that describes what action to trigger, and a *condition* part that defines the conditions for triggering the alarm. The general format for an alarm is as follows:

**Action** | **Condition**

**@action**
The symbolic name of an alarm. Alarm names must start with '@' and otherwise contain only alphanumeric characters.

**alarm_definition**
One or more of "[command line]", "{TRAPxx}", and "{EXCEPTION}".

**bool_expression**
{evariable|wildcard|const} (boolean_operator {evariable|wildcard|const}) ...

**boolean_operator**
An operator which evaluates an expression to produce a value of true or false. Supported operators:

```
 '='  Equal
"!="  Not Equal
'>'   Greater Than
'<'   Less Than
">="  Greater Than or Equal
"<='  Less Than or Equal
'&'   AND
'|'   OR
```

**const**
[digits]

**description**
Text describing the alarm. must be enclosed in double quotation marks. The text cannot be more than 512 bytes in length.

**evariable**
Fully qualified **xmperf** variable name with slashes replaced by underscores. Valid names have at least one underscore. The first name component must start with an alpha character, subsequent ones may also begin with a percent sign. All must contain only alpha-numeric characters, a percent sign, a tilde (~), a period, or an underscore preceded by an escape character (the backslash '\', or a wildcard. The referenced variable may be defined by this same filter, in which case it must be specified as: DDS_IBM_Filters_target, where "target" is the name of the new statistic.

**wildcard**
A fully qualified **xmperf** variable name with slashes replaced by underscores. Valid names have at least one underscore. The first name component must start with an alpha character, subsequent ones may also begin with a percent sign. All must contain only alpha-numeric characters and percent signs or must be a wildcard. The wildcard character must appear in place of a context name must only appear once and must be one of the characters '+', '*', '#', '>', '<' .

## Alarm Definition

An alarm can define up to three actions to take place when the alarm condition is met. These three actions are:

**[command line]** A command line to be executed when the alarm condition is met. The command line must be enclosed in square brackets. The command line is always executed in the background and with the same credentials as that of the **filtd** daemon. If the **filtd** daemon has been started by **xmservd**, the command line is executed with root authority.

**{TRAPxx}** This action can always be specified but it only produces the desired results if the **xmservd** daemon is configured to export its statistics to the **snmpd** daemon through the xmservd/SMUX interface described in Chapter 15, “SNMP Multiplex Interface,” on page 177.

**Note:** The interface to SMUX is only available on RS/6000 Agents.

If **xmservd** does talk to the **snmpd** daemon, this type of action will, when the defined condition becomes true, produce an SNMP trap that is passed on through **xmservd** to **snmpd** and, eventually, to an SNMP manager such as NetView. The keyword **TRAP** must be in uppercase letters and must be followed by one or more decimal digits defining the trap number. Both the keyword and the trap number must be enclosed in curly braces. The trap sent to **snmpd** is an enterprise-specific trap (generic type 6) with a specific trap number equal to the number specified after the **TRAP** keyword.

**{EXCEPTION}** This action causes the **filtd** daemon to inform the **xmservd** daemon each time the defined condition is met. This makes **xmservd** send a message of type **except_rec** to all hosts having declared that they want to receive such messages. “Requesting Exception Messages” on page 25 explains how a data-consumer program can request to be informed about exceptions. The message contains the identification, description and other data from the alarm definition. The exact layout of the message is declared in the file **/usr/include/sys/Spmidef.h** as **Exception_Rec** and is included in the union of message types in file **/usr/include/sys/Rsi.h**.

## Alarm Duration and Frequency

The two keywords **DURATION** and **FREQUENCY** are used to determine how long time a condition must remain

### true

```
           Default    Minimum
          ----------  ---------
DURATION   60 seconds  1 second
FREQUENCY  30 minutes  1 minute
```

For an alarm to be triggered, at least **FREQUENCY** minutes must have elapsed since the last time this same alarm was triggered. When this is the case, the condition is monitored constantly. Each time the condition switches from false to true, a time stamp is taken. As long as the condition stays true, the elapsed time since the last time stamp is compared to **DURATION** and, if it equals or exceeds **DURATION**, the alarm is triggered.

When it can be done without forcing the data feed interval to become less than one second, **filtd** makes sure at least three data feeds will be taken in **DURATION** seconds. This is done by modifying the data feed interval, if necessary. Doing this can have side effects on new statistics you have defined, since there’s only one data feed interval in use for all raw statistics received by the **filtd** program, whether the raw statistics are used to define new statistics, to define alarms, or both.

## Alarm Severity

A severity code can be associated with an alarm. This is intended to be used when you define one of the actions that result from an alarm shall be to send an **except_rec** to a data-consumer program. Unfortunately, there’s no way to associate a severity level with an SNMP trap.

If you do not specify a severity code, a default of 1 is used. Severity can currently be specified as a value from 0 to 10. The higher the value, the more severe the alarm.

## Examples of Alarm Definitions

Alarms need not really be alarms. It would be much nicer if the conditions that would usually trigger an alarm could cause corrective action to be taken without human intervention. One example of such corrective action is that of increasing the UDP receive buffers in case of UDP overrun. You could do this with the following “alarm” definition:

```
@udpfull:[no -o sb_max=262144] UDP_fullsock > 5 DURATION 1
```

If you wanted an SNMP trap with specific number 31 to be sent in addition to the execution of the **no** command, you would define the alarm as:

```
@udpfull:[no -o sb_max=262144]{TRAP31} UDP_fullsock > 5 DURATION 1 \
        "Another UDP buffer overrun"
```

If you wanted to be informed whenever the paging space on your host has less than 10 percent free space or there’s less than 100 pages free paging space, you could use an alarm definition like the following:

Our final example defines an alarm to send an **except_rec** to interested data-consumer programs whenever the average busy percent for the disks exceeds 50 for more than 5 seconds:

```
@diskbusy:{EXCEPTION} (Disk_+_busy) / (Disk_#_busy) > 50 DURATION 5 \
          SEVERITY 3 "Disks are more than 50% busy on average"
```

## Using Raw Values and Delta Values

“Quantities and Counters” on page 185 explains the consequences of using counter values when you define new statistics. For an **SiCounter** statistic, the **filtd** daemon always assumes that you are referring to the rate per second if you specify the path name of the statistic with no suffix. This means that of the two value fields in a **data_feed** packet, the one that contains the delta value is used and divided by the time interval covered by the packet to arrive at the rate.

If this is not what you want, one of two available suffixes can be added to the path name to take the corresponding value and not divide it with the time interval. Those two suffixes are:

**@Raw**
: When you use this suffix, the value used to construct the new statistic is the raw counter value of the counter if the statistic is of type **SiCounter**. If this suffix is used for statistics of type **SiQuantity** you will see no difference from not using a suffix.

**@Delta**
: When used for statistics of type **SiQuantity**, the value used to construct the new statistic is undefined. Don’t use this suffix for quantities. When used for **SiCounter** type statistics, the value used to construct the new statistic is the delta value as it appears in the **data_feed** packet. The value is not divided by the time interval.

The suffixes do not change the anomalies explained in “Quantities and Counters” on page 185. They are available because the raw data values or the delta values may be useful in other contexts. It is strongly suggested that any use of the suffixes is thoroughly tested before the results are made available to end users.

To illustrate the use of the suffixes, a few examples follow. The first example shows how to define a new statistic that contains the change in the counter of ticks in user mode:

```
userdelt = CPU_cpu0_uticks@Delta
```

The value of `userdelt` is the change of the counter value over the time interval. If the sampling interval is 5 seconds, the value will be approximately five times the average percent user CPU over the interval. If you want to see the absolute counter value for the number of ticks in kernel mode, type the following code:

```
userkern = CPU_cpu0_kticks@Raw
```

The final example defines one new statistic and an alarm that will be triggered when the counter of CPU idle ticks wraps. It looks like this:

```
idleraw = CPU_cpu0_iticks@Raw
@wrap:{EXCEPTION} CPU_cpu0_iticks@Raw dds_ibm_filters_idleraw \ severity 8
duration 1 "idle counter wrapped"
```

The trick here is that the statistic **idleraw** is one cycle behind the statistic from which it is derived. Therefore, a wrap can be detected as shown.

# Chapter 17. Response Time Measurement

This chapter provides information about the response time measurement facilities of the Performance Toolbox for AIX (PTX) and the Performance Aide for AIX. Except where otherwise noted, the facilities described are available on all platforms supported by the Performance Aide. Monitoring of response times across the network can be done from workstations only.

## Introduction

Response time measurement is especially important in a client/server environment and is ideally done on a transaction basis. The problem is that a transaction is an elusive concept. Between client and server, transactions may range from causing a single network transmission with no response to involving a large number of transmissions. In any one customer installation, one or a few typical transactions may be found, and the selected transactions can then be instrumented (possibly through the implementation of the *Application Response Measurement API* **(ARM)** described in "Application Response Time Measurement (ARM)" on page 195. Using the response time measurement of a few transaction types representing a large percentage of the actual transactions performed, it is possible to get a feel for the responsiveness of all or most transaction types.

Transaction instrumentation is the most precise vehicle for response time measurement but for this concept to work, the installation must be willing to invest in the analysis of transaction patterns and instrumentation of transaction programs. The installation must also be *able* to modify the transaction programs, which is not always possible. For example, how does one instrument a standard SQL query program? Because it is expensive, somewhat complex, and often impossible to use the transaction instrumentation concept, other means must be used in an attempt to monitor system responsiveness.

Those other means involve the measurement of the atomic components that, together, add up to the response time of a given transaction. The following is a list of some major steps in a client/server application. Each is followed by some resources that are required by the task and, hence, will influence the response time component if they are scarce.

1. Client application processes user input (CPU, disk)
2. Client machine enqueues network request (CPU, adapter, network)
3. Request is transferred over network (network capacity and speed)
4. Server enqueues request (CPU, adapter)
5. Server application processes request (CPU, disk, possibly access to other servers)
6. Server machine enqueues response(CPU, adapter, network)
7. Response is transferred over network (network capacity and speed)
8. Client application processes response (CPU, disk, possibly access to other servers)
9. Client application sends response to end-user (CPU, terminal network).

All of the resources in the previous list can be monitored by PTX when it comes to activity counts. Disks can be monitored for the percent of time they are busy, which give a good feel for their responsiveness, but networks can be monitored only for the activity counts. Furthermore, while activity counts for disks can be used to judge how close a disk is to being saturated, the activity counts for one machine's network adapter may have little or no connection to the actual load on the network. Maybe one machine is constantly accessing remote files while another seldom is. The low activity count on the second machine is no guarantee for a fast response when the machine does need remote access.

It only makes the situation worse that, in a typical client/server application, the largest response time component is usually the time is takes to get the request sent and the response back. That's why the IP response time measurement facility was added to PTX. IP response time measurement works by using the low level Internet Control Management Protocol (ICMP) to send responses to selected hosts and

© Copyright IBM Corp. 1994, 2004

measuring the time it takes to get a response back. The ICMP protocol was chosen because it doesn't require an application to be running on the remote host, because the protocol is handled by the IP implementation itself.

## IP Response Time Measurement

In PTX, IP response time measurement is implemented through a daemon and corresponding contexts in the Spmi data hierarchy. The Spmi will start the daemon as required and will dynamically add contexts for all the remote hosts, for which monitoring is started.

### IP Response Time Daemon

Measuring of response times is done by a daemon called **SpmiResp**. If this daemon is not running when the Spmi receives a request for IP response time measurements, it is started by the Spmi library code. The daemon will continue to run until it has been the only user of the Spmi interface for 60 seconds or no data consumer has requested response time data for 300 seconds. When running, the **SpmiResp** daemon is controlled by an interval timer loop. The interval timer is, by default, set to interrupt the daemon every 10 seconds but the *interval* value can be changed from the daemon's configuration file **/etc/perf/Resptime.cf**.

Whenever the daemon is interrupted by the timer, it starts a new cycle, sending one ICMP packet to each host for which response time is being monitored and calculating the response time from the time it takes for the response to come back. The daemon will not attempt to send more frequently than specified by a variable maxrate ("Configuring the SpmiResp Daemon" on page 193 section), which defaults to 10 packets per second but can be changed from the daemon's configuration file **/etc/perf/Resptime.cf**. When sending packets, the daemon will attempt to spread its activity evenly over the interval ("Configuring the SpmiResp Daemon" on page 193 section) seconds a cycle lasts. If the number of monitored hosts is too large for all hosts to be contacted within *interval* seconds without exceeding *maxrate*, then interruptions by the interval timer are ignored until a full cycle has been completed. The reason for having the *maxrate* parameter is to prevent the measurement of network activity from being distorted by burst of ICMP packets from the response time monitor.

When a response is received to an ICMP package, the response time is calculated as a fixed point value in milliseconds. In addition. the weighted average response time is calculated as a floating point value using a variable weight ( "Configuring the SpmiResp Daemon" on page 193 section), that defaults to 75%. The average response time is calculated as *weight* percent of the previous value of the average plus (100 - *weight*) percent of the latest response time observation. The value of *weight* can also be changed from the daemon's configuration file **/etc/perf/Resptime.cf**.

### IP Response Time Metrics

The following metrics are maintained. Except where noted, all values are floating point values:

**resptime**
The latest observed response time in milliseconds (fixed point value).

**respavg**
The weighed average response time in milliseconds.

**below10**
The percentage of observations of response time that were less than 10 milliseconds.

**below20**
The percentage of observations of response time that were less than 20 milliseconds but greater than 10 milliseconds.

**below100**
The percentage of observations of response time that were less than 100 milliseconds but greater than 20 milliseconds.

**above99**
The percentage of observations of response time that were at 100 or more milliseconds.

**requests**
A counter value giving the number of ICMP requests sent to the host (fixed point value).

**responses**
A counter value giving the number of ICMP responses received from the host (fixed point value).

## Configuring the SpmiResp Daemon

The **SpmiResp** daemon looks for a configuration file in **/etc/perf/Resptime.cf**. Three values can be specified in this file. A keyword identifies which value is being set. The keyword must appear in column one of a line and white space must separate the keyword and the value. The three values, as identified by the corresponding keywords are:

**interval**
The interval in seconds between each loop of **SpmiResp**. Default is 10 second intervals.

**maxrate**
The maximum rate **SpmiResp** will send ICMP packets with; packets per second. Default is 10 packets per second.

**weight**
The weight a previous value has in finding the weighted average of the response time. Default is 75%.

If no configuration file is found, **SpmiResp** continues with default control values. The detailed meaning and use of the three values is described in "IP Response Time Daemon" on page 192.

The daemon will catch all major signals. All, with the exception of SIGHUP, will cause the daemon to shut gracefully down. SIGHUP will cause the daemon to reread its configuration file. Any value specified in the configuration file will replace whichever corresponding value is currently active.

## IP Response Time Contexts

For applications to be able to monitor response time through the Spmi and, ultimately, the RSi interface, the data must be available in context and metric data structures in the Spmi shared memory area. However, it would consume many resources to create these data structures for all hosts in a large network, so the Spmi has been modified to handle response time contexts different from all other context types. What happens is that a context for a particular host is not created until some consumer of data refers to that context by its path name.

For a data consumer program to see the IP response time context for a host, either some other program must have created the context, or the program itself must attempt to get to the context through the context's path name. For example, to create (or access if somebody else created it) the context for the host farvel, the application could issue the following call:

```
SpmiPathGetCx("RTime/LAN/farvel", NULL);
```

The same effect is achieved by issuing the **RSiPathGetCx** subroutine call, which ultimately leads to an **SpmiPathGetCx** subroutine call on the agent host the RSi call is issued against.

This implementation leaves the Data Consumer applications with the responsibility of identifying hosts to monitor for IP response time. This is contrary to all other contexts in the Spmi, which can be instantiated simply by traversing their parent contexts. If an installation wants to make sure all IP response time contexts are created, the sample Data Consumer program in **/usr/samples/perfagent/server/iphosts.c**, which is also shipped as the executable **iphosts**, can be executed from the **xmservd.res** file whenever **xmservd** starts. This program will take a file with a list of hostnames as input and will issue the **SpmiPathGetCx** call for each host.

Because IP response time measurement uses the **ICMP** protocol, the hosts you want to monitor do not need to run the **xmservd** daemon. All that is required is that they can respond properly to ICMP echo requests. Because of this, response time to *any* node that talks **ICMP**, including dedicated routers and gateways, can be measured.

Two PTX applications use their knowledge of hosts in the network to present their users with lists of potential IP response time contexts. The two are **xmperf** and **3dmon**. This is described in "Monitoring IP Response Time from xmperf" on page 194 and "Monitoring IP Response Time from 3dmon" on page 194.

# Monitoring IP Response Time from xmperf

The **xmperf** program presents its user with potential IP response time contexts in two situations:

1. When the user displays the value selection window for the context **RTime/LAN**.
2. When a user instantiates a skeleton console that refers to IP response time measurement.

In both cases, the list of potential IP response time contexts includes the hosts whose **xmservd** daemons responded to invitations from **xmperf**, as well as hosts for which contexts were added by other means such as by the **iphosts** program. If the host is known by two names (typically by the short hostname and the full host/domain name) the same host may appear twice in the list. The same may happen if the **iphosts** program identifies hosts by their IP addresses rather than their hostnames.

## Host List in Value Selection Window

When the user wants to add a statistic to an instrument or exchange one statistic value with another, the value selection windows are used to select the new statistic. If the user selects **RTime** from the top value selection window and from the next window selects **LAN**, then the new value selection window displayed will show one context for each host that can be monitored. By selecting a host, you automatically create the context for measurement of response time for that host if the context doesn't exist. You then proceed to the value selection window for that context.

## Instantiating an IP Response Time Skeleton Console

When a user instantiates an IP response time measurement skeleton console, the list of hosts to select from looks like any other host selection list, except that the list may contain lines that do not show an IP address. Such lines represent hosts for which an IP response time context exists but that have not responded to the **xmperf** invitation. If instantiation is tried again after the host list has been refreshed, more hosts may have IP addresses shown. This reflects how the host list is created. If the same hostname is available from both the list of hosts that responded to invitations and from existing IP response time contexts, then the entry from the list of responding hosts is used. Also, if a host is a little late responding to an invitation, it may first show up without its IP address. After the response to the invitation finally is received, it will show up with its IP address.

## Remounting an IP Response Time Monitor

It is possible to create **xmperf** instruments that monitor response times for multiple hosts. This illustrates that even though it looks like the instrument receives **data_feed** packets from multiple hosts, in reality it receives packets from only one host. All values in an instrument must come from the same host and must be defined in the same *statset*. The Spmi on the measurement host receives ICMP responses (not data feeds) from the monitored hosts and has been instructed to supply the calculated response time data in statsets.

The looks of an instrument measurement IP response time for multiple hosts and of a console with multiple instruments, each monitoring IP response time for multiple hosts can be deceiving, though. You might be tempted to change the path of an instrument or the entire console, expecting to be able to select a new list of hosts to monitor. You will get a list of hosts from which you can select one host; that will then be the new *monitoring* (not *monitored*) host.

# Monitoring IP Response Time from 3dmon

The **3dmon** program takes a different approach to monitoring IP response time. It does so in a matrix that monitors the response time in both directions. For example, if you elect to monitor response times for three hosts, `trist`, `ked`, and `nede`, then the matrix would look like this:

```
                    NN

               KN              NK

        TN             KK              NT

nede           TK              KT              nede
```

ked TT ked

trist trist

The right side of the matrix represents the *monitoring* hosts, and the left side represents the *monitored* hosts. Thus, the value represented in the cell marked `NT` is the response time for a response to an ICMP echo packet sent from host `nede` to host `trist`. The response time for a response to an ICMP echo packet sent in the other direction is shown in the cell marked `TN`. This measuring of response time in both directions allows for the detection of situations where two hosts use different routes to reach each other and it can also be used to pinpoint other anomalies in a network.

Because of the way **3dmon** monitors IP response time, all of the hosts must be running the **xmservd** daemon. Therefore, when **3dmon** shows a list of hosts to monitor IP response times for, only hosts that responded to invitations from **3dmon** are included. The distributed **3dmon.cf** sample configuration file includes a configuration set named **lanresp** for monitoring of IP network response time.

## Application Response Time Measurement (ARM)

This section describes the Performance Aide and Performance Toolbox implementation of the *Application Response Measurement API* **(ARM)**. To see where ARM would be useful, revisit the following list of common steps a client/server transaction goes through:

1. Client application processes user input (CPU, disk)
2. Client machine enqueues network request (CPU, adapter, network)
3. Request is transferred over network (network capacity and speed)
4. Server enqueues request (CPU, adapter)
5. Server application processes request (CPU, disk, possibly access to other servers)
6. Server machine enqueues response (CPU, adapter, network)
7. Response is transferred over network (network capacity and speed)
8. Client application processes response (CPU, disk, possibly access to other servers)
9. Client application sends response to end-user (CPU, terminal network)

An application can be instrumented at many levels. For example, one set of measurements could cover the entire period from the beginning of step 1 to the end of step 9. Another, potentially simultaneous, set of measurements could cover the server side beginning with step 4 or 5 and ending with step 6. The ARM API, in its current, unfinished state, depends on the instrumentation to supply meaningful names to applications and transactions. Without such names, measurement would be impossible.

## ARM Contexts in Spmi Data Space

As implemented in PTX, the ARM support is limited by the design of ARM as done by the original designers. The two-level naming structure of PTX, where every context and metric has a short name and a long descriptive name was not implemented in ARM. This puts a single requirement on how an application is instrumented. The first 31 bytes of the description of an application and a transaction must uniquely define that application and transaction within the application.

Applications are added to the Spmi data hierarchy as contexts. An application with the description “Checking Account Query” will be added as the context:

```
RTime/ARM/CheckingAccountQuery
```

A transaction of that application called “Last Check Query” would be added to the previous context as another context level and get a full path name as follows:

```
RTime/ARM/CheckingAccountQuery/LastCheckQuery
```

If the transaction was named “Check if any check of this account has bounced during the past 12 months”, the full path name of the transaction context would be:

```
RTime/ARM/CheckingAccountQuery/Checkifanycheckofthisaccounthas
```

With the short name being truncated to 31 characters. The full name of the transaction would appear in the description of the transaction context, truncated to 63 characters if necessary.

## ARM Transaction Metrics

For each transaction, the following metrics are maintained. All metrics, with the exception of **respavg** are fixed point values:

**resptime** The last measured response time for successful transaction in milliseconds.

**respavg** The weighted average response time for successful transactions in milliseconds.

**count** The number of successful transactions.

**aborted** The number of aborted transactions.

**failed** The number of failed transaction.

**respmax** The maximum response time for successful transactions in milliseconds.

**respmin** The minimum response time for successful transactions in milliseconds.

To reduce the memory usage, do not attempt to determine the median response time or the 80-percentile.

## Implementation Restrictions

The ARM API is not implemented by Performance Aide on the SunOS operating system.

At this point, the **arm_update** subroutine is implemented as a null function. This is because the current monitors of PTX wouldn’t be able to monitor transaction progress in a well-defined manner. This may change in a future version of PTX. Other implementation restrictions are listed under each API function.

For the **SpmiArmd** daemon to get write access to the Spmi common shared memory, the daemon must be started with root or system group authority. The safest way to make this happen is to make sure the **xmservd** daemon is always running. This can be accomplished by entering the flag **-l0** (lowercase L followed by a zero) to the server’s line in **/etc/inetd.conf**. It is also recommended that the following line is added to the appropriate **xmservd.res** file, which is used to start data suppliers:

```
supplier: /usr/bin/SpmiArmd
```

Because the PTX implementation uses shared memory, and because library code cannot feasibly catch all relevant signals, application instrumentors must make sure that an **arm_end** call is issued for each active application. If a program exits while applications it defined are still active, the shared memory area will not be released and the **SpmiArmd** daemon will assume that data is still being supplied and will not attempt to exit. This is not likely to be a major draw-back but if things get tricky, it may be necessary to stop the daemon (and all other programs using the Spmi) and clear shared memory manually as described in Releasing Shared Memory Manually.

## Library Implementation

The ARM specifications prescribe that the ARM library is shipped as a shared library such as **libarm.a** or **libarm.so**. Replacing the installed library with another library with the same interfaces will redirect application subroutine calls to the library installed last. The implementation of ARM in PTX follows these specifications but also allows a customer installation to invoke both an existing ARM library and the PTX implementation of ARM. This is achieved by shipping the following two libraries:

**/usr/lib/libarm.a** A plain ARM implementation, which does not invoke any pre-existing ARM implementation.

**/usr/lib/libarm2.a** A replacement library for the plain library.

To use the replacement library, convert the preexisting ARM library by running the following command:

```
armtoleg /usr/lib/libarm.a /usr/lib/libarmrepl.a >
/dev/null
```

Then copy **/usr/lib/libarm2.a** over **/usr/lib/libarm.a**. The replacement library will invoke the ARM functions in **/usr/lib/libarmrepl.a** before invoking the PTX ARM implementation in the replacement library. This way, a customer installation can continue to use an earlier ARM instrumentation and at the same time take advantage of the PTX implementation of ARM. When the replacement library is used, the behavior of the PTX ARM implementation changes but remains compliant with the ARM specifications. Further information on these subroutines are in the *AIX 5L Version 5.3 Technical Reference*.

Both PTX libraries depend on the **SpmiResp** daemon to be started by the Spmi library code. This daemon must run with root authority in order to interface directly with the Spmi common shared memory area. The daemon is described in the "SpmiArmd Daemon" section.

The library code maintains state in a private shared memory area, which can be accessed with any authority, thus allowing applications to do so through the library calls. The library communicates with the Spmi through the **SpmiArmd** daemon, which issues standard Spmi subroutine calls. No direct communication takes place between the Spmi and the ARM library.

## Run-time Control

The use of two environment variables allows an application to turn both levels of ARM instrumentation on and off. Because they are environment variables, they work for the shell from which the application is executed and have no effect on the execution from other shells. The two environment variables are as follows:

**INVOKE_ARM**
: Controls the PTX ARM instrumentation. If the environment variable is not defined or if it has any value other than `False`, PTX ARM instrumentation is active. If the replacement library is not used, the effect of setting this environment variable to `False` is that the PTX ARM library will function as a no-operation library and return zero on all calls.

**INVOKE_ARMPREV**
: Controls any ARM instrumentation that can be invoked from the PTX implementation through the replacement library. If the environment variable is not defined or if it has any value other than False, the preexisting ARM instrumentation will be invoked, regardless of whether the PTX ARM instrumentation is active. If the replacement library is not used, this environment variable has no effect.

If both environment variables are set to `False`, either PTX ARM library will function as a no-operation library and return zero on all calls.

## SpmiArmd Daemon

Collection of application response times is done by a daemon called **SpmiArmd**. If this daemon is not running when the Spmi receives an **SpmiGetCx**, see the **SpmiGetCx** subroutine call referencing ARM contexts, it is started by the Spmi. The daemon will continue to run until it has been the only user of the Spmi interface for 15 minutes and no data has been received from an instrumented application for 15 minutes. The time to live can be changed from the daemon's configuration file **/etc/perf/SpmiArmd.cf**. When running, the **SpmiArmd** daemon is controlled by an interval timer loop. The interval timer is, by

default, set to interrupt the daemon every second but the interval (see **interval** in the“Configuring the SpmiArmd Daemon” section table) value can be changed from the daemon’s configuration file **/etc/perf/SpmiArmd.cf**.

Whenever the daemon is interrupted by the timer, it empties the entire queue of *post structures* with the exception of the last entry (see **ATake**, in the “SpmiArmd Daemon” on page 197 section table), using each to update the corresponding metrics. The metrics are updated as follows:

1. If the post element indicates that the transaction instance completed successfully, response time is calculated. The response time is calculated as a fixed point value in milliseconds. In addition, the weighted average response time is calculated as a floating point value using a variable weight (“Configuring the SpmiArmd Daemon” section), that defaults to 75%. The average response time is calculated as *weight* percent of the previous value of the average plus `(100 - weight)` percent of the latest response time observation. The value of *weight* can be changed from the **SpmiArmd** daemon’s configuration file **/etc/perf/SpmiArmd.cf**. In addition, the maximum and minimum response time for this transaction is updated, if required. Finally the counter of successful transaction executions is incremented.
2. If the post element doesn’t indicate a successful execution, either the aborted or failed counters are incremented. No other updates occur.

To eliminate the daemon from the need to lock data in the ARM library’s private shared memory area, the following technique is used to control the linked list of post structures. The following three fields in the shared memory control area are used as anchors:

**ATake**
: Points to the first post element to be processed by the daemon. After initializing, this field is never updated by the library code. The daemon reads elements starting at **ATake** and processes them. It stops when the next element has a next pointer of NULL and then sets **ATake** to point at that element.

**AGive**
: Points to an uninitialized post element and is used by the library code to add new post elements. When the shared memory area is first initialized, both **ATake** and **AGive** point to an empty element. As new post elements are needed, the library code allocates them or takes them from the **AFreePost** list. The element pointed to by **AGive** is then updated. The last step in updating this element is the setting of its next pointer to the newly acquired element, which will have a NULL next pointer.

**AFreePost**
: Points to a linked list of post elements. The elements between this pointer and **ATake** are unused elements and will be reused by the library code. This field is never updated by the daemon. Whenever an element is taken off this list, the **AFreePost** anchor is updated to point at the next element. Initially, this anchor is set equal to **AGive** an **ATake**.

The daemon catches all viable signals and will exit for all but SIGHUP. When the daemon receives a SIGHUP signal, it rereads the configuration file and re-initializes its control variables to any new values specified in the configuration file.

## Configuring the SpmiArmd Daemon

The **SpmiArmd** daemon looks for a configuration file in **/etc/perf/SpmiArmd.cf**. Three values can be specified in this file. A keyword identifies which value is being set. The keyword must appear in column one of a line and white space must separate the keyword and the value. The three values, as identified by the corresponding keywords are as follows:

**interval**
: The interval in seconds between each loop of **SpmiArmd**. Default is 1 second.

**weight**
: The weight a previous value has in finding the weighted average of the response time. Default is 75%.

**timeout**
: The number of seconds the daemon should live with no activity going on. The default is 900 seconds (15 minutes). A value of zero for this parameter will cause the daemon to live forever.

If no configuration file is found, **SpmiArmd** continues with default control values.

## Monitoring ARM Metrics from xmperf

The **xmperf** program presents its user with ARM contexts in the following two situations:

1. When the user displays the value selection window for the context RTime/ARM.
2. When a user instantiates a skeleton console that refers to measurement of ARM metrics.

### ARM Context List in Value Selection Window

When the user wants to add a statistic to an instrument or exchange one statistic value with another, the value selection windows are used to select the new statistic. If the user selects **RTime** from the top value selection window and from the next window selects **ARM**, then the new value selection window displayed will show one context for each application that can be monitored. By selecting an application you are taken to the next selection window, which will present a list of the transactions defined for the application. By selecting a transaction context you proceed to the value selection window for that application/transaction context.

### Instantiating ARM Skeleton Console

ARM skeleton consoles must be defined for each application you want to monitor because **xmperf** doesn't support dual wildcards (as does **3dmon**). When a user instantiates an ARM skeleton console, a list of transactions within the application is presented for the user to select from. Each line represent an application/transaction context. One or multiple lines can be selected.

### Remounting an ARM Monitor

It is not possible to create **xmperf** instruments that monitor ARM data for multiple hosts. However, consoles can be constructed with instruments that each monitor a different host. ARM instruments can be remounted on different hosts as any other instrument.

## Monitoring ARM Metrics from 3dmon

The **3dmon** program permits wildcarding in multiple levels. This allows you to create configurations corresponding to those available for file systems. You can chose to be presented first with a list of hosts and then by a list of all application/transaction combinations defined for that host. You can also restrict the list to the application/transaction combinations for a single host.

The resulting **3dmon** display will show the host/application/transaction name on the left side and the name of configured metrics on the right side. A configuration set named **armresp** for monitoring of application response time is available in the distributed **3dmon.cf** configuration file.

## Sample Applications

Source code for an instrumented application is not supplied with PTX but a modified version of the **xmpeek** program is shipped as **/usr/samples/perfagent/server/armpeek**. This program is an instrumented version of the ordinary **xmpeek**. It creates an application called *armpeek* and one transaction for each combination of command line flag and hostname. For example, the command **armpeek -l myhost** will create and measure a transaction called *myhost-l*; the command **armpeek** refers to the local host and would create a transaction called *localhost*; the command **armpeek -a server** would create the transaction *server-a*.

# Chapter 18. System Performance Measurement Interface Programming Guide

The System Performance Measurement Interface (SPMI) is an application programming interface (API) that provides standardized access to local system resource statistics. By developing SPMI application programs, a user can retrieve information about system performance with minimum system overhead.

## SPMI Overview

Two types of application programs can use the SPMI:

- Data user applications that use the API to access SPMI data structures
- Dynamic Data Supplier (DDS) applications that use the API to add statistics and data structures to those already available from the SPMI.

*AIX 5L Version 5.3 Technical Reference: Communications Volume 2* must be installed to see the SPMI subroutines.

The following figure illustrates the two different application types using the SPMI:

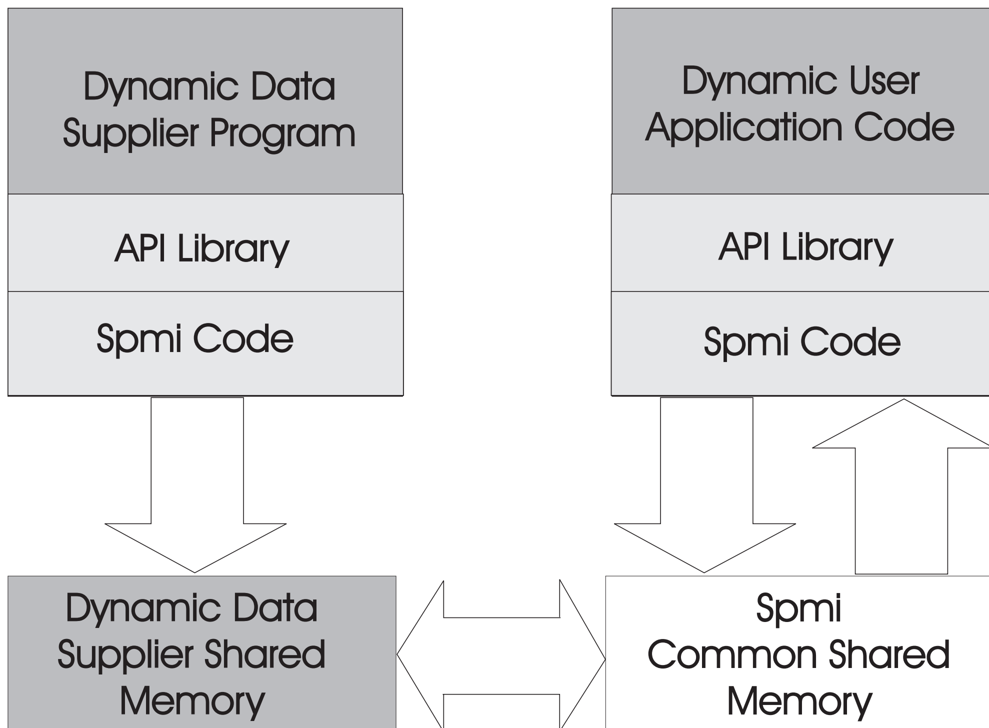



*Figure 7. Application Types. This illustration shows the DDS program on the left and the Data User Application Code on the right. Both are connected to an API library and SPMI code. The DDS program column has one way communication with the Dynamic Data Supplier Shared Memory that has two-way communication with the SPMI Common Shared Memory. The Dynamic Data Supplier Shared Memory has two-way communication with the SPMI Common Shared Memory.*

## Possible Uses for the SPMI

The SPMI assists programmers with the following tasks:

- Developing performance monitoring products.

- Retrieving performance statistics using applications created with minimal programming and without programming at the kernel level.
- Exporting an application's performance statistics. The exporting application would be a dynamic data supplier. A performance monitoring program could access these statistics without requiring changes to the performance monitoring program.

## SPMI Features

The SPMI offers the following features:

**Counter Data**
Enables access of activity rate statistics, such as the number of disk-read operations per second.

**Level Data**
Enables access of system-usage statistics, such as the usage level of real memory.

**Single Data Repository**
Enables multiple data-user application programs, running simultaneously, to access the same set of statistics.

**Self Declarative Data**
Enables programs to dynamically present a list of available statistics to a user. The SPMI arranges all performance data in a hierarchy, with related statistics grouped together. The hierarchy contains context nodes (called *contexts*) and leaf nodes (called *statistics*). Each context may have subordinate contexts, statistics, or both. A program can traverse the data hierarchy through simple get and get-next calls to the SPMI API. Each context and statistic has a short name and a descriptive name.

**Instantiation**
Enables the SPMI to monitor multiple copies of a system resource. A context (and its subcontexts and statistics) may exist in multiple instances, such as for each disk, each CPU, or each active process.

**Expandable Interface**
Enables a DDS application program to expand the set of statistics without affecting the API.

**Note:** Versions earlier than 2.3 of the SPMI in Performance Toolbox for AIX do not allow simultaneous access of SPMI subroutines from multiple threads of a process.

## Understanding the SPMI Data Hierarchy

SPMI data is organized in a multilevel hierarchy of *contexts*. A context may have subordinate contexts, known as *subcontexts*, as well as statistics. The higher-level context is called a *parent* context.

The following figure illustrates a data hierarchy for a multiprocessor system. Each ellipse depicts a context or subcontext, and each rectangle depicts a statistic. The CPU context consists of a subcontext for each of the processors. The Top context serves as an anchor point for all other contexts.

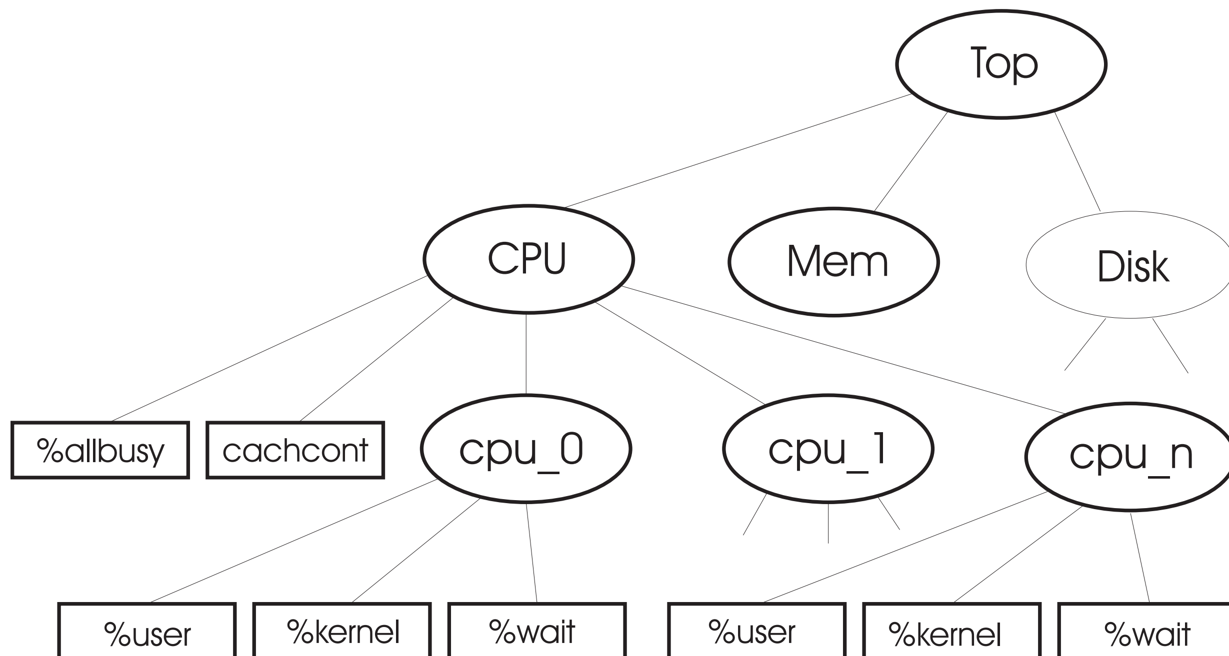



*Figure 8. Sample Data Hierarchy. This illustration depicts several ellipses and rectangles. Each ellipse depicts a context or subcontext; CPU, Memory, Disk and so on. Each rectangle depicts a statistic; %user, %kernel, %wait, and so on.. The CPU context consists of a subcontext for each of the processors. The Top context serves as an anchor point for all other contexts. If, while looking for a statistic called %kernel, a SPMI application program follows the CPU context, that program finds the following instances;* `CPU/cpu_0/%kernel`*, or* `CPU/cpu_1/%kernel`*, or* `CPU/cpu_n/%kernel`*.*

If a SPMI application program (while looking for a statistic called %kernel), follows the CPU context (shown in the previous figure), that program finds the following instances:

```
CPU/cpu_0/%kernel
CPU/cpu_1/%kernel
. . .
CPU/cpu_n/%kernel
```

## Instantiation

When multiple copies of a resource are available, the SPMI uses a base context description as a template. The SPMI creates one instance of that context for each copy of the resource or system object. This process is known as *instantiation*. The previous figure illustrates instantiation by showing multiple copies of the CPU context used to describe the processors. A context is considered instantiable if at least one of its immediate subcontexts can exist in more than one copy.

The SPMI can generate new instances of the subcontexts of instantiable contexts prior to the execution of API subroutines that traverse the data hierarchy. An application program can also request instantiation explicitly. In either case, instantiation is accomplished by requesting the instantiation for the parent context of the instances. For example, to instantiate all processor contexts in the previous figure, the application would request instantiation at the CPU context.

Some instantiable contexts always generate a fixed number of subcontext instances in a given system, as long as the system configuration remains unchanged. Examples of this type of instantiability are contexts that contain subcontexts for network interface cards or processors.

Some contexts generate a fixed number of subcontexts on one system, but not on another. Using the Disk context as an example, if System A has a fixed number of disks while System B has removable disk drives, the number of subcontexts for the Disk context of System B changes as disk drives are added and removed while the number of subcontexts for System A is constant.

A final type of context is entirely dynamic in that it will add and delete instances as required during operation. For example, process subcontexts are repeatedly added and deleted during usual system operation.

## Instantiability

Because an application program can request instantiation for any context, the contexts are defined as having one of three types of instantiability:

**Not instantiable**
: The context contains either no subcontexts or a fixed number of subcontexts. Most contexts that are themselves subcontexts of an instantiable context belong to this group. In the Sample Data Hierarchy figure, the Top context and the cpu_0 through cpu_n subcontexts are not instantiable because none of their immediate subcontexts can occur in more than one copy. The Mem context is also not instantiable, assuming the system processors share one contiguous bank of memory.

**Instantiable at system-configuration time**
: The number of instances never changes unless the system is reconfigured. In the Sample Data Hierarchy figure, the CPU context is an example of this kind of instantiability because the number of processors is constant.

**Dynamically instantiable**
: The actual number of instances can vary during system operation. Subcontexts need not be of the same context type. For example, one dynamically instantiable context may have a group of subcontexts with an instance for each active socket and another group with an instance for each Transmission Control Protocol (TCP) connection. In such an arrangement, the number of instances of both groups is likely to change quite frequently in a network system.

# Understanding SPMI Data Areas

The SPMI uses a shared memory segment created from user space. When an SPMI application program starts, the SPMI checks whether another program has already set up the SPMI data structures in shared memory. If the SPMI does not find the shared memory area, it creates one and generates and initializes all data structures. If the SPMI finds the shared memory area, it bypasses the initialization process. A counter, called users, shows the number of processes currently using the SPMI.

When an application program terminates, the SPMI releases all memory allocated for the application and decrements the users counter. If the counter drops to less than 1, the entire common shared memory area is freed. Subsequent execution of an SPMI application reallocates the common shared memory area.

The **sys/Spmidef.h** file contains declarations of all the SPMI data structures.

## Traversing the Data Hierarchy

An application program has access to the data hierarchy through the API. The following figure gives a simplified picture of the underlying data structures. A set of subroutines allows the application program to navigate through the structures, reviewing what data is available. This action is known as *traversing* the data hierarchy. The traversal process allows a user to find statistics of interest. To extract data from these statistics, an application program must define a set of statistics, called a *statset*. See “Data Access Structures and Handles, StatSets” on page 208 for additional information.

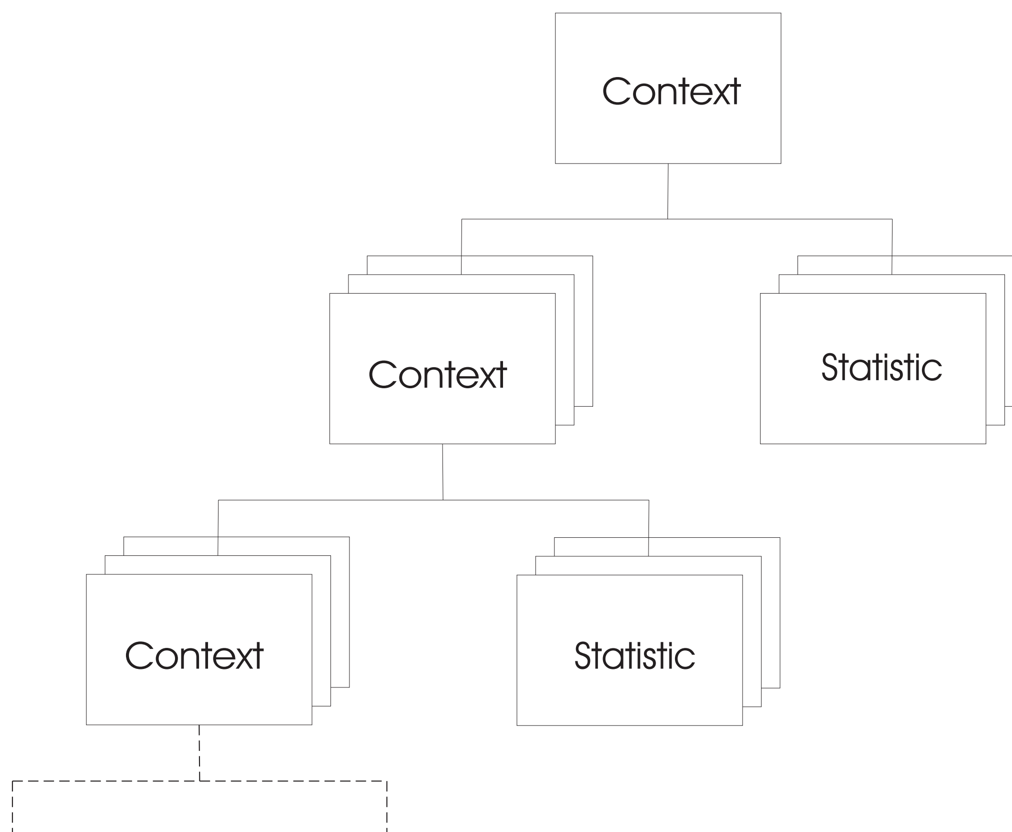



*Figure 9. Application Program View of Data Hierarchy. This illustration shows a hierarchy or organizational-type chart with context at the top. It has a set of contexts and a set of statistics branching off from it. Each set of contexts has another set of contexts and a set of statistics branching off from it. The dashed lines at the bottom signify the continuing set of contexts and statistics that follow all sets of contexts.*

## Data Traversal Structures and Handles

To traverse the data hierarchy, an application program uses four data structures, two handles, and a set of subroutines. The structures include:

- “Declaring a Context - the cx_create Structure” on page 214
- “SpmiCx Structure” on page 206
- “SpmiCxHdl Handle” on page 206
- “SpmiCxLink Structure” on page 207
- “SpmiHotItems” on page 211
- “SpmiHotSet Structure” on page 210
- “SpmiHotVals Structure” on page 210
- “Declaring a Statistic - the SpmiRawStat Structure” on page 213
- “SpmiStat Structure” on page 206
- “SpmiStatHdl Handle” on page 207
- “SpmiStatLink Structure” on page 207
- “SpmiStatSet Structure” on page 208
- “SpmiStatVals Structure” on page 209.

## SpmiCx Structure

The **SpmiCx** structure describes a context node in the data hierarchy. As seen by an application program, an **SpmiCx** data structure is always an instance of a context. The data structure names and describes the context in the name and description fields, respectively. The structure also contains a symbolic reference, called a *handle*, for accessing the parent context (this handle is NULL if the parent context is Top) and a field describing the instantiability of the context. The asnno field contains an Abstract Syntax Notation One (ASN.1) number that makes the structure unique. See “Making Dynamic Data-Supplier Statistics Unique” on page 214 for more information about ASN.1.

The **SpmiCx** structure is defined as follows:

```
#define SI_MAXNAME 32
#define SI_MAXLNAME 64

enum SiInstFreq
{
   SiNoInst,      /* Subcontexts never change */
   SiCfgInst,     /* Subcontext changes are system configuration changes */
   SiContInst,    /* System operation changes subcontexts continuously */
};
struct SpmiCx
{
   char            name[SI_MAXNAME];         /* short name of the context   */
   char            description[SI_MAXLNAME]; /* descriptive name            */
   SpmiCxHdl       parent;                   /* handle of parent context    */
   enum SiInstFreq inst_freq;                /* instantiability of context  */
   u_short         asnno;                    /* ASN.1 number                */
   u_char          deleted;                  /* nonzero if context deleted  */
   u_char          dummy;                    /* alignment                   */
};
```

## SpmiCxHdl Handle

The **SpmiCxHdl** handle is a symbolic reference to a context. To access the **SpmiCx** structure identified by the handle, use the **SpmiGetCx** subroutine as follows:

```
struct SpmiCx         *spmicx;
SpmiCxHdl             cxhdl;

spmicx = SpmiGetCx(cxhdl);
```

## SpmiStat Structure

The **SpmiStat** structure describes a *statistics object*. The object always describes a single-field counter or level statistic. Typically, a system component updates these fields and the update process is asynchronous to any requests to read the field. The field itself is not contained in the **SpmiStat** structure, but SPMI subroutines allow an application program to retrieve the field value.

The **SpmiStat** structure, like the **SpmiCx** structure, associates a name and a description with each object. It also defines default scale values by anticipating the low and high ranges of the field. For a counter field, the **SpmiStat** structure defines the low and high ranges of the change anticipated for a 1-second time period (the event rate per second).

The **ValType enum** defines the **SiCounter** and **SiQuantity** used to describe the data type as either a counter or a level, respectively. The **SpmiStat** structure also contains the format of the field. Though the **enum DataType** field defines many field formats, only **SiLong** and **SiFloat** are currently supported.

The **asnno** field contains an abstract syntax notation number that makes the structure unique from other structures. See “Making Dynamic Data-Supplier Statistics Unique” on page 214 for more information.

The **SpmiStat** structure is defined as follows:

```
#define SI_MAXNAME 32
#define SI_MAXLNAME 64
```

```
enum ValType
{
   SiCounter,   /* field is always incremented */
   SiQuantity,  /* field maintains a level     */
};
enum DataType
{
   SiULong,
   SiLong,
   SiUInt,
   SiInt,
   SiUShort,
   SiShort,
   SiChar,
   SiAddr,
   SiTimeval,
   SiFloat,
   SiDouble,
   SiPtr,
   SiUnsign,
};
struct SpmiStat
{
   char           name[SI_MAXNAME];         /* short name of statistic     */
   char           description[SI_MAXLNAME]; /* descriptive name            */
   long           min;                      /* default low scale value     */
   long           max;                      /* default high scale value    */
   enum ValType   value_type;               /* data type presented to API  */
   enum DataType  data_type;                /* data format presented to API*/
   u_short        asnno;                    /* ASN.1 number                */
   u_short        dummy;                    /* alignment                   */
};
```

## SpmiCxLink Structure

The “SpmiFirstCx Subroutine” on page 340 and **SpmiNextCx** subroutines use the **SpmiCxLink** structure to traverse the subcontexts of a context. The **Spmi**CxLink structure serves as a handle when passed as a parameter to the **SpmiNextCx** subroutine. The structure contains both a reserved field and a field that is the handle of the subcontext.

The **SpmiCxLink** structure is defined as follows:

```
struct SpmiCxLink
{
   void       *reserved;      /* reserved field, don't change */
   SpmiCxHdl  context;        /* handle of subcontext         */
};
```

## SpmiStatLink Structure

The **SpmiFirstStat** and **SpmiNextStat** subroutines use the **SpmiStatLink** structure to traverse the statistics of a context. This structure serves as a handle when passed as a parameter to the **SpmiNextStat** subroutine. The structure contains both a reserved field and a field that is the handle of the statistic.

The **SpmiStatLink** structure is defined as follows:

```
struct SpmiStatLink
{
   void        *reserved;     /* reserved field, don't change */
   SpmiStatHdl stat;          /* handle of statistic          */
};
```

## SpmiStatHdl Handle

The **SpmiStatHdl** handle is a symbolic reference to a statistic. To access the **SpmiStat** structure identified by the handle, use the **SpmiGetStat** subroutine as follows:

```
struct SpmiStat   *spmistat;
SpmiStatHdl       stathdl;

spmistat = SpmiGetStat(stathdl);
```

## Data Access Structures and Handles, StatSets

To access data values, the application program must define the data values it needs to the SPMI. For this purpose, the application program defines sets of statistics, or statsets, through the API. A set of statistics is anchored to a data structure defined as an **SpmiStatSet** structure. The address of a defined **SpmiStatSet** structure must be passed to the SPMI through the API each time the application program needs to access the actual data values referenced by the structure.

When the SPMI receives a read request for an **SpmiStatSet** structure, the SPMI returns the latest value for all the statistics in the set of statistics. This action reduces the system overhead caused by access of kernel structures and other system areas, and ensures that all data values for the statistics within a set are read at the same time. The set of statistics may consist of one or many statistics fields.

The SPMI builds internal data structures for the set in response to API calls from the application program. The following figure illustrates a simplified look at the way the application views these structures through the API.

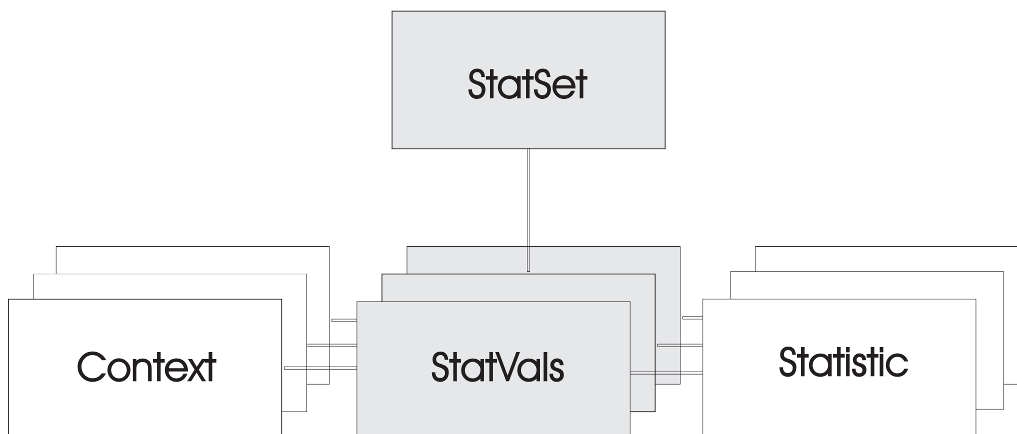



*Figure 10. Data Value Access Structures. This illustration shows an organization chart with a rectangle for StatSet at the top linking to a set of StatVal rectangles that individually link to contexts and statistics.*

One “SpmiStatVals Structure” on page 209 is created for each of the data values selected for the set. When the SPMI executes a request from the application program to read the data values for a set, all **SpmiStatVals** structures in the set are updated. The application program can then either traverse the list of **SpmiStatVals** structures, by using the **SpmiFirstVals** and **SpmiNextVals** subroutines, or retrieve single values by using the **SpmiGetValue** or **SpmiGetNextValue** subroutines.

An application program uses the following data structures to create, delete, and access sets of statistics.

### SpmiStatSet Structure

This structure is an anchor point for the structures that define a set of statistics. The application program is responsible for creating the **SpmiStatSet** structure, adding and deleting statistics, and deleting the **SpmiStatSet** structure when no longer needed. The SPMI depends on the application program to supply the address of an **SpmiStatSet** structure. The structure holds only the time stamp for the most recent reading of its associated statistics and the elapsed time since the previous reading.

The **SpmiStatSet** structure is defined as follows:

```
struct SpmiStatSet
{
   struct timeval    time;              /* time of current get        */
   struct timeval    time_change;       /* elapsed time since last get */
};
```

### SpmiStatVals Structure

The **SpmiStatVals** structure carries the data values from the SPMI to the application program. It contains handles that allow an application to access the parent context and the **SpmiStat** structure of the value. The ref_count field indicates the number of times a particular statistic is included in the same set. The ref_count field usually has a value of 1.

The **SpmiStatVals** structure is defined as follows:

```
union Value
{
   long    l;
   float   f;
};

struct SpmiStatVals
{
   void             *reserved;         /* reserved field             */
   SpmiStatHdl      stat;              /* handle of statistic        */
   SpmiCxHdl        context;           /* handle of context          */
   int              ref_count;         /* count of simultaneous users */
   union Value      val;               /* counter/level data value    */
   union Value      val_change;        /* delta change if counter data*/
   enum Error       error;             /* error code                 */
};
```

The last three fields actually transfer the data values, as follows:

**val**
Returns the value of the counter or level field. This field returns the statistic's value as maintained by the original supplier of the value. However, the val field is converted to an SPMI data format.

**val_change**
Returns the difference between the previous reading of the counter and the current reading when the statistic contains counter data. When this value is divided by the elapsed time returned in the**SpmiStatSet** structure, an event rate-per-time unit can be calculated.

**error**
Returns a zero value if the SPMI's last attempt to read a data value was successful. Otherwise, this field contains an error code as defined in the**sys/Spmidef.h** file. See the "List of SPMI Error Codes" on page 242 for more information.

## Data Access Structures and Handles, HotSets

To access data values, the application program can define a different type of StatSet to the SPMI. It is used to extract data values for the most or least active statistics for a group of peer contexts. For example, it can be used to define that the program wants to receive information about the two highest loaded disks, optionally subject to those values exceeding a specified threshold. For this purpose, the application program defines sets of peer statistics, called hotsets, through the API.

A hotset is anchored to a data structure defined as an **"SpmiHotSet Structure" on page 210**. The address of a defined **SpmiHotSet** structure must be passed to the SPMI through the API each time the application program needs to access the actual data values referenced by the structure.

When the SPMI receives a read request for an **SpmiHotSet** structure, the SPMI reads the latest value for all the peer sets of statistics in the hotset in one operation. This action reduces the system overhead caused by access of kernel structures and other system areas, and ensures that all data values for the peer sets of statistics within a hotset are read at the same time. The hotset may consist of one or many sets of peer statistics.

The SPMI builds internal data structures for the set in response to API calls from the application program. Figure 10 on page 208 illustrates the statset access structures. The structures for the hotset look exactly the same but the referenced context is now the parent context of all peers to examine. This parent context is what groups the peers together. All of the subcontexts of that parent context are of the same type and are considered peer contexts. By naming a specific statistic for one peer context, you reference the same statistic for each of the peer contexts, hence the term *a set of peer statistics.*

One "SpmiHotVals Structure" structure is created for each set of peer statistics selected for the hotset. When the SPMI executes a request from the application program to read the data values for a hotset, all **SpmiHotVals** structures in the set are updated. The application program can then either traverse the list of **SpmiStatVals** structures, by using the **SpmiFirstHot** and **SpmiNextHot** subroutines, or retrieve single values by using the **SpmiNextHotItem** subroutine.

An application program uses the following data structures to create, delete, and access hotsets.

## SpmiHotSet Structure

This structure is an anchor point for the structures that define a group of peer statistic sets. The application program is responsible for creating the **SpmiHotSet** structure, adding and deleting peer sets of statistics, and deleting the **SpmiHotSet** structure when no longer needed. The SPMI depends on the application program to supply the address of an **SpmiHotSet** structure. The structure holds only the time stamp for the most recent reading of its associated statistics and the elapsed time since the previous reading.

The **SpmiHotSet** structure is defined as follows:

```
struct SpmiHotSet
{
   struct timeval    time;              /* time of current get           */
   struct timeval    time_change;       /* elapsed time since last get   */
};
```

## SpmiHotVals Structure

The **SpmiHotVals** structure carries the data values from the SPMI to the application program. It contains handles that allow an application to access the parent context of the peer contexts and the **SpmiStat** structure of the peer statistic. The ref_count field indicates the number of times a particular set of peer statistics is included in the same set. The ref_count field usually has a value of 1.

The **SpmiHotVals** structure is defined as follows:

```
union Value
{
   long    l;
   float   f;
};

enum HotExcept {
   SiHotNoException = 0,
   SiHotException,
   SiHotTrap,
   SiHotBoth
};

enum HotFeed {
   SiHotNoFeed = 0,
   SiHotThreshold,
   SiHotAlways
};

struct SpmiHotItems
{
   char             name[SI_MAXLNAME]; /* name of the peer context    */
   union Value      val;               /* counter/level data value    */
   union Value      val_change;        /* delta change if counter data*/
};
```

```
struct SpmiHotVals
{
   void              *reserved;        /* reserved field              */
   SpmiStatHdl       stat;             /* handle of statistic         */
   SpmiCxHdl         grandpa;          /* parent of the peer contexts */
   int               ref_count;        /* count of simultaneous users */
   enum Error        error;            /* error code                  */
   enum HotExcept    except_type;      /* when to send exceptions     */
   short             trap_no;          /* trap number for SNMP traps  */
   short             severity;         /* severity for exception pckt */
   enum HotFeed      feed_type;        /* when to send data feeds     */
   int               threshold;        /* threshold for what to send  */
   short             frequency;        /* max frequency of exceptions */
   short             max_responses;    /* max  # of responses to send */
   short             avail_resp;       /* # of available hot readings */
   short             count;            /* # of returned hot readings  */
   char              *path;            /* path to grandpa context     */
   struct SpmiHotItems *items;         /* array of returned readings  */
};
```

The data carrying fields are:

**error**
: Returns a zero value if the SPMI's last attempt to read the data values for a set of peer statistics was successful. Otherwise, this field contains an error code as defined in the **sys/Spmidef.h** file. See the "List of SPMI Error Codes" on page 242 for more information.

**avail_resp**
: Used to return the number of peer statistic data values that meet the selection criteria (threshold). The field **max_responses** determines the maximum number of entries actually returned.

**count**
: Contains the number of elements returned in the array **items**. This number will be the number of data values that met the selection criteria (threshold), capped at **max_responses**.

**items**
: The array used to return **count** elements. This array is defined in the **SpmiHotItems** data structure.

### SpmiHotItems

An array of this structure is pointed to by the field **items** in the **SpmiHotVals** structure. Elements are ordered after the returned data values; ascending if **threshold** is negative, otherwise descending. Each element in the array has the following fields, used to return the result. Note that each instance of this structure corresponds to one single peek statistic within a set of peer statistics as defined in an instance of an **SpmiHotVals** structure.

**name**
: The name of the peer context for which the values are returned.

**val**
: Returns the value of the counter or level field for the peer statistic. This field returns the statistic's value as maintained by the original supplier of the value. However, the **val** field is converted to an SPMI data format.

**val_change**
: Returns the difference between the previous reading of the counter and the current reading when the statistic contains counter data. When this value is divided by the elapsed time returned in the **SpmiStatSet** structure, an event rate-per-time-unit can be calculated.

## Dynamic Data Supplier (DDS) Program Structures

A DDS program can register and supply private statistics to the SPMI by calling SPMI subroutines. The following information describes the program structures used to communicate between a DDS program and the SPMI.

A DDS program initializes the API by using the "SpmiDdsInit Subroutine" on page 335 subroutine. This subroutine allocates the DDS shared memory area, shown in "Data-Supplier Shared Memory Layout" on page 212, based on the two data structures described in:

- "Declaring a Statistic - the SpmiRawStat Structure" on page 213
- "Declaring a Context - the cx_create Structure" on page 214

DDS programs should not directly manipulate the shared memory area, its control information, or its data structures except by using the following fields:

**SiShGoAway**
Indicates that the DDS program is no longer needed or that its data is corrupted. This field may be set by any data-consumer program. Usually when a DDS program sees that this field has true value, the program calls the **SpmiExit** subroutine to free the allocated shared memory and unlink from the SPMI interface. Failure to call the **SpmiExit** subroutine before exiting retains the allocated shared memory. As a result, the DDS program cannot be restarted until the shared memory is freed using the **ipcrm** command.

**SiShT**
Specifies a time stamp that must be updated by the DDS program each time the shared data area is updated. Data-consumer programs may check this field to see when the DDS program was last active. If too much time elapses without a time-stamp update, the data-consumer program may assume the DDS is inoperative, and will set the **SiShGoAway** field and release its access to the shared memory area.

**Note:** The time stamp is a structure with two integer elements. It is expected to be stored in Big Endian notation. Hosts that use Little Endian notation must convert the integer fields to Big Endian notation before storing them.

**SiShArea**
DDS programs must use the **SiShArea** field. This field contains the address of the data area in the shared memory segment. A DDS program must load a pointer with this field and use that pointer to access the shared memory data area. The program can do calculations directly in the area allocated in shared memory or do the calculations in local data fields and then move the results to shared memory.

## Data-Supplier Shared Memory Layout

The following structure shows the DDS shared memory layout:

```
typedef struct
{
   short              SiShMajor;     /* Major version of shm protocol    */
   short              SiShMinor;     /* Minor version of shm protocol    */
   char               SiShName[64];  /* Path name for shm allocation     */
   char               SiShId;        /* ID for ftok() function           */
   key_t              SiShKey;       /* shared memory key (RSi interface)*/
                                     /* creating process ID (Spmi I/F)   */
   int                SiShMemId;     /* shared memory identifier         */
   u_long             SiShInetAddr;  /* IP address of owning host        */
   u_short            SiShPortNo;    /* port number to talk to daemon    */
   int                SiShAllocLen;  /* length of allocated area         */
   int                SiShInstBegun; /* instantiations begun             */
   int                SiShInstDone;  /* instantiations completed         */
   int                SiShRefrBegun; /* refreshes begun                  */
   int                SiShRefrDone;  /* refreshes completed              */
   boolean            SiShGoAway;    /* signal supplier to terminate     */
   boolean            SiShAlarmSw;   /* switch to indicate alarm is set  */
   cx_create          *SiShCxTab;    /* pointer to fixed context table   */
   int                SiShCxCnt;     /* count of contexts in above table */
   cx_create          *SiShInstTab;  /* pointer instantiable contexts    */
   int                SiShInstCnt;   /* count of contexts in above table */
   struct SpmiRawStat *SiShStatTab;  /* pointer to consolidated Stats    */
   int                SiShStatCnt;   /* count of Stats in above table    */
   char               *SiShArea;     /* pointer to statistics area       */
   int                SiShAreaLen;   /* length of statistics area        */
   struct timeval     SiShT;         /* time of last area update BY US   */
   struct timeval     SiShPost;      /* time of update of fields below   */
   int                SiShInterval;  /* sample frequency in milliseconds */
   int                SiShSubscrib;  /* current number of values used    */
   struct SpmiCxLink  *SiShAddCx;    /* instantiated contexts to add     */
   struct SpmiCxLink  *SiShActCx;    /* active instantiated contexts     */
   struct SpmiCxLink  *SiShDelCx;    /* contexts to delete               */
   struct SpmiCxLink  *SiShFreeCx;   /* freed contexts                   */
   void               *SiShAlarm;    /* addr of Shm alarm data area      */
   u_long             SiShLock1;     /* lock words to serialize access   */
```

```
   u_long            SiShLock2;     /*  .. from multiple threads/CPUs   */
   u_long            SiShRes[4];    /* reserved for future use          */
   char              SiShData;      /* start of data area               */
} SpmiShare;
```

## Declaring a Statistic - the SpmiRawStat Structure

To add permanent statistics, a DDS program must describe the statistics in the **SpmiRawStat** structure. For each context with statistics that the program wants to add, the program must create a table of statistics.

The **SpmiRawStat** structure is defined as follows:

```
#define SI_MAXNAME 32
#define SI_MAXLNAME 64

enum ValType
{
   SiCounter,   /* field is always incremented */
   SiQuantity,  /* field maintains a level     */
};

enum DataType
{
   SiULong,
   SiLong,
   SiUInt,
   SiInt,
   SiUShort,
   SiShort,
   SiChar,
   SiAddr,
   SiTimeval,
   SiFloat,
   SiDouble,
   SiPtr,
   SiUnsign,
};

struct SpmiRawStat
{
   char             name[SI_MAXNAME];  /* short name of statistic     */
   char             description[SI_MAXLNAME]; /* descriptive name     */
   long             min;               /* default low scale value     */
   long             max;               /* default high scale value    */
   enum ValType     value_type;        /* data type presented to API  */
   enum DataType    data_type;         /* data format presented to API*/
   u_short          asnno;             /* ASN.1 number                */
   u_short          size;              /* source data field size      */
   int              offset;            /* source data field offset    */
   enum DataType    type;              /* source data field format    */
   int              (*get_fun)();      /* data access function pointer*/
#ifdef _SOLARIS
   int              ksnoffs;           /* Solaris ksn_struct offset   */
   char             module[SI_MODL];   /* Solaris kstat source module */
   char             statname[SI_STAL]; /* Solaris kstat stat name     */
   char             fieldname[SI_FLDL];/* Solaris kstat field name    */
   int              datoffs;           /* Solaris ksn data offset     */
#endif /* _SOLARIS */
};
```

Application programs should leave the section inside **#ifdef _SOLARIS** uninitialized, even on Solaris systems. The following example defines the “gadgets” and “widgets” statistics. See “Example of an SPMI Dynamic Data-Supplier Program” on page 237 for a sample program that uses this definition.

```
static CONST struct SpmiRawStat PUStats[] = {
{  "gadgets", "Fake counter value", 0, 100, SiCounter,
   SiLong, 1, SZ_OFF(dat, a, SiULong), NULL},
```

```
{  "widgets", "Another fake counter value", 0, 100,
SiCounter,
   SiLong, 2, SZ_OFF(dat, b, SiULong), NULL},
};
```

### Declaring a Context - the cx_create Structure

After declaring the statistics, the DDS program must link them to their parent contexts. To do so, the program uses a single table of structures that defines all the contexts as permanent contexts. Each context requires one element of the **cx_create** type.

The **cx_create** structure is defined as follows:

```
#define SI_MAXNAME 32
#define SI_MAXLNAME 64

typedef struct
{
   char           path[SI_MAXLNAME];   /* context path name              */
   char           descr[SI_MAXLNAME];  /* context description            */
   u_short        asnno;               /* ASN.1 number                   */
   u_short        datasize;            /* size of context record         */
   struct SpmiRawStat   *stats;        /* Stat array pointer for context */
   int            num_stats;           /* element count of Stat array    */
   struct SpmiRawStat   *inst_stats;   /* Stat array for multiple        */
                                       /*   instances of this context    */
   int            num_inst_stats;      /* element count for above array  */
   int            (*inst_subs)();      /* function to instantiate context*/
   int            inst_freq;           /* instantiate frequency          */
   u_long         level;               /* relative level (work field)    */
   char           *area;               /* data area pointer              */
   u_long         arealen;             /* length of above data area      */
} cx_create;
```

The following example defines the **DDS/IBM** and **DDS/IBM/sample1** contexts. See "Example of an SPMI Dynamic Data-Supplier Program" on page 237 for a sample program that uses this definition.

```
static CONST cx_create cx_table[] = {
   {"DDS/IBM", "IBM-defined Dynamic Data Suppliers", 2, 0,
      NULL, 0, NULL, 0, NULL, SiNoInst},
   {"DDS/IBM/sample1", "Bogus Context Number 1", 191, 0,
      PUStats, STAT_L(PUStats), NULL, 0, NULL, SiNoInst},
};
```

## Making Dynamic Data-Supplier Statistics Unique

Some structures contain an asnno field. The SPMI assigns each context and statistic a unique number in Abstract Syntax Notation One (ASN.1) format and stores the number in this field. As a result, the SPMI can export SPMI and DDS statistics to other interfaces, such as the Simple Network Management Protocol (SNMP). ASN.1 identifiers are composed of a series of integers separated by dots. That is, the context called CPU has the ASN.1 identifier of 1. The subcontexts of CPU have ASN.1 identifiers that are numbered starting with 1. The ASN.1 identifiers for statistics belonging to each subcontext of CPU are also numbered starting with 1. For instance, for the **CPU/cpu0/idle** path name the **CPU** context has an ASN.1 identifier of 1, the **CPU/cpu0** subcontext has an ASN.1 identifier of 1.1, and the **CPU/cpu0/idle** statistic has an ASN.1 identifier of 1.1.4. This identifier is referred to as the *relative* dotted-decimal identifier.

The dotted-decimal identifiers of all statistics can be thought of as a substructure or subtree, which can be attached to any point in another network, such as the SNMP Management Information Base (MIB) tree. For example, an SNMP tree has a point defined as the following:

```
internet.private.enterprises.ibm.ibmAgents.aix.risc6000.risc6000private
```

This subtree is graphically illustrated in the following figure.

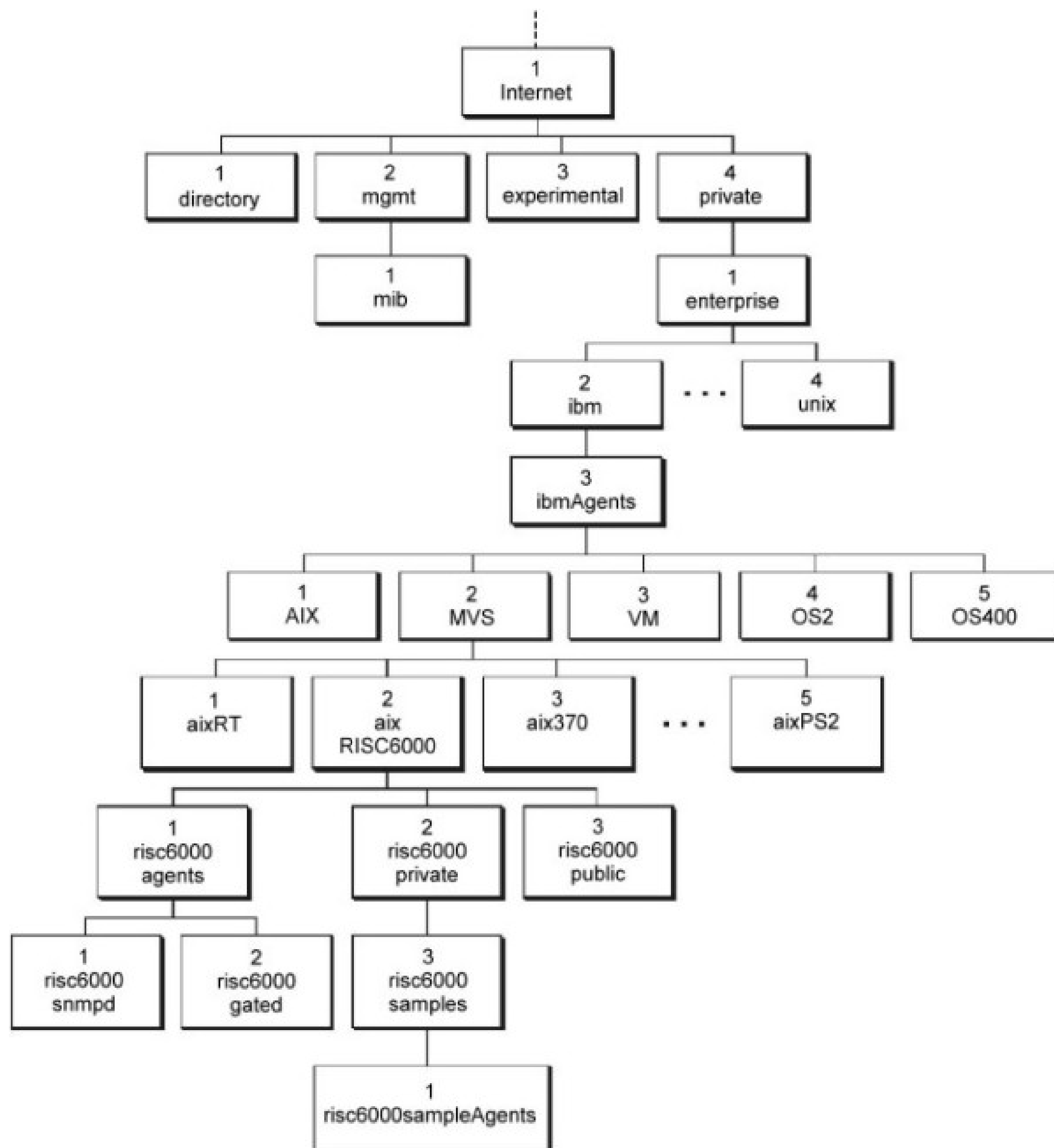



*Figure 11. Hierarchical Organization of Private MIB Subtrees. This diagram shows eight layers within the subtree. Private is below internet. Enterprises branches off private and so on, as the previous point is defined. (The items within the same layer and stemming from the same base are numbered from left to right.)*

This description corresponds to a dotted-decimal identifier of 1.3.6.1.4.1.2.3.1.2.2 (the numbers 1.3.6.1 correspond to the internet portion of the tree). If the SPMI contexts were attached at this point, the fully qualified dotted-decimal identifier for the **CPU/cpu0/idle** path name (1.1.4) would be 1.3.6.1.4.1.2.3.1.2.2.1.1.4.

No two contexts or statistics can have the same relative dotted-decimal identifier. The API checks that you do not assign the same ASN.1 number more than once at each level in the subtree defined by your DDS program. This ensures unique relative dotted-decimal identifiers.

In addition, statistics and contexts must have unique names within the parent context. The API checks that DDS programs adhere to this rule so that the full path name of statistics and contexts remains unique.

However, the API can only detect name or ASN.1 number clashes, not resolve them. Therefore, it is recommended that DDS programs use the following naming and numbering scheme. This scheme uses an SPMI-defined context, called DDS, with a relative dotted-decimal identifier of 99. When adding subtrees, DDS programs should use the DDS context as the parent context.

This naming and numbering scheme is graphically illustrated in the following figure.

Vendors of DDS programs should define a private subcontext of the DDS context as the parent context for all DDS subtrees that the program creates. Assign this private subcontext a meaningful name and an ASN.1 number that corresponds to the ASN.1 number assigned to the vendor in the SNMP subtree **enterprises**. Such a number is called an Assigned Enterprise Number. For example, IBM has an Assigned Enterprise Number of 2. Therefore, the subcontext for the path name **DDS/IBM** would have a relative dotted-decimal identifier of 99.2.

DDS programs can be loaded in any sequence. However, because all DDS programs depend on the presence of the vendor-specific context (such as **DDS/IBM**), each program must attempt to add this context to make sure it exists. After a program has created the vendor-specific context, it remains defined as long as the common shared memory area exists. When other DDS programs attempt to add the same vendor-specific context, they simply use the already created context. Vendor-specific contexts must never have statistics defined. Statistics should be added to subcontexts of the vendor-specific context.

**Note:** Vendors without SNMP numbers should register for one by contacting the Internet Assigned Numbers Authority at the following address:

```
Internet Assigned Numbers Authority
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, California 90202-6695
```

The e-mail address for the Internet Assigned Numbers Authority is **iana@isi.edu**.

## Using the System Performance Measurement Interface API

The API supplied with the Agent component is called the **System Performance Measurement Interface (SPMI)**. It allows you to write programs that extend the number of statistics available from a host's **xmservd** daemon (dynamic data-supplier programs) and to write programs that access statistics on the local host without using the network interface (local data-consumer programs).

This chapter describes how you use this API to create your own dynamic data-supplier program. The sample programs used to explain the API and several additional ones are provided in machine-readable form as part of the Agent component. The sample programs and a Makefile can be found in directory:

**/usr/samples/perfagent/server**

## Using SPMI to Create a Dynamic Data Supplier

When you want to extend the set of statistics available from the **xmservd** daemon on a host, you create a dynamic data-supplier (DDS) program using the SPMI API. When the DDS program executes, it registers its statistics with the SPMI. This makes the new statistics immediately available to the local data-consumer programs and to the **xmservd** daemon so any program that gets its statistics from the extended **xmservd** daemon can access the additional statistics provided by the DDS. DDS programs must execute on the same host as the one running the **xmservd** whose set of statistics is to be extended.

## Makefiles

The include files are based upon a number of pre-processor define directives being properly set. They must be defined with the **-D** preprocessor flag.

- *_AIX* Tells the include files to generate code for any version of the operating system.

- *_AIX_*41 Tells the include files to generate code for AIX 4.1 and AIX 4.2 of the operating system.
- *_AIX_*32 Tells the include files to generate code for AIX 3.2 of the operating system.
- *_BSD* Required for proper BSD compatibility.

A Makefile to build all the sample programs provided could look like the following:

```
LIBS = -lbsd -lSpmi
CC = cc
CFLAGS = -D_BSD -D_AIX-D_AIX_41

all:: SpmiDds SpmiSupl SpmiSupl1 SpmiLogger SpmiPeek lchmon
lfiltd

SpmiDds: SpmiDds.c
  $(CC) -o SpmiDds SpmiDds.c $(CFLAGS) $(LIBS)

SpmiSupl: SpmiSupl.c
  $(CC) -o SpmiSupl SpmiSupl.c $(CFLAGS) $(LIBS)

SpmiSupl1: SpmiSupl1.c
  $(CC) -o SpmiSupl1 SpmiSupl1.c $(CFLAGS) $(LIBS)

SpmiLogger: SpmiLogger.c
  $(CC) -o SpmiLogger SpmiLogger.c $(CFLAGS) $(LIBS)

SpmiPeek: SpmiPeek.c
  $(CC) -o SpmiPeek SpmiPeek.c $(CFLAGS) $(LIBS)

lchmon: lchmon.c  $(CC) -o lchmon lchmon.c $(CFLAGS) $(LIBS)
 -lcurses

lfiltd: lfiltd.c lex.lfiltd.o lfiltd.h
  $(CC) -o lfiltd lfiltd.c lex.lfiltd.o $(CFLAGS) $(LIBS)

lex.lfiltd.o: lfiltd.lex lfiltd.h
  lex lfiltd.lex
  cp lex.yy.c lex.lfiltd.c
  rm lex.yy.c
  $(CC) -c lex.lfiltd.c $(CFLAGS)
```

To compile on non-AIX systems, other flags must be used. A Makefile is included with each non-AIX agent. Please use flags as defined in that Makefile. If the compiler you are using doesn't support ANSI function prototyping, add the flag:

```
-D_NO_PROTO
```

## Writing Dynamic Data-Supplier Programs

A dynamic data-supplier program is intended to extend the set of statistics that data-consumer programs can be supplied with, either from the **xmservd** daemon of a host or directly from the SPMI repository through local data-consumer programs. A dynamic data-supplier can add statistics as permanent (non-volatile) or dynamic (volatile) contexts with subcontexts and statistics. To illustrate this concept, assume the SPMI has a set of contexts and statistics as illustrated in the following figure.

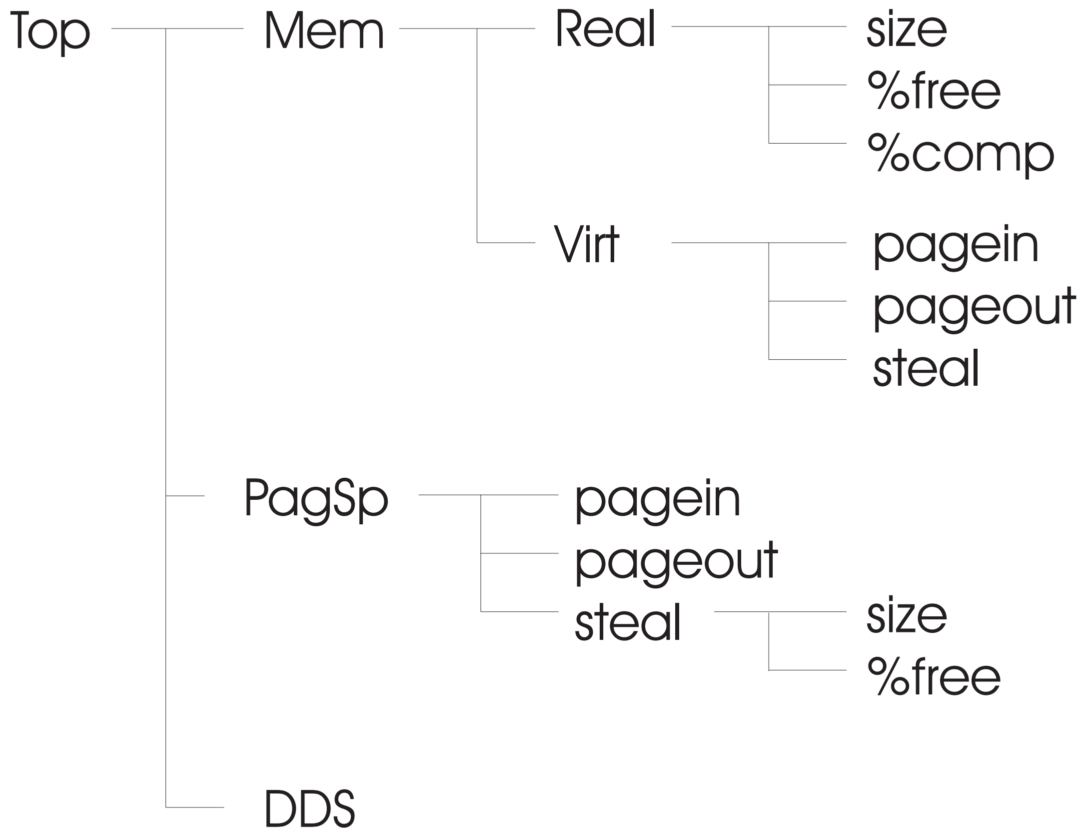



*Figure 12. Start Set of Statistics. This figure shows a set of parent contexts named Mem, PagSp, and DDS. The first contexts have subcontexts representing samples of those found on the operating system.*

The set of statistics on an IBM RS/6000 is much larger than shown, so this is used as an illustration. Now assume that you have access to other statistics and want them added to the set. This is when you want to create a dynamic data-supplier program. For example, you could extend the tree structure of contexts and statistics to look as shown in the following figure.

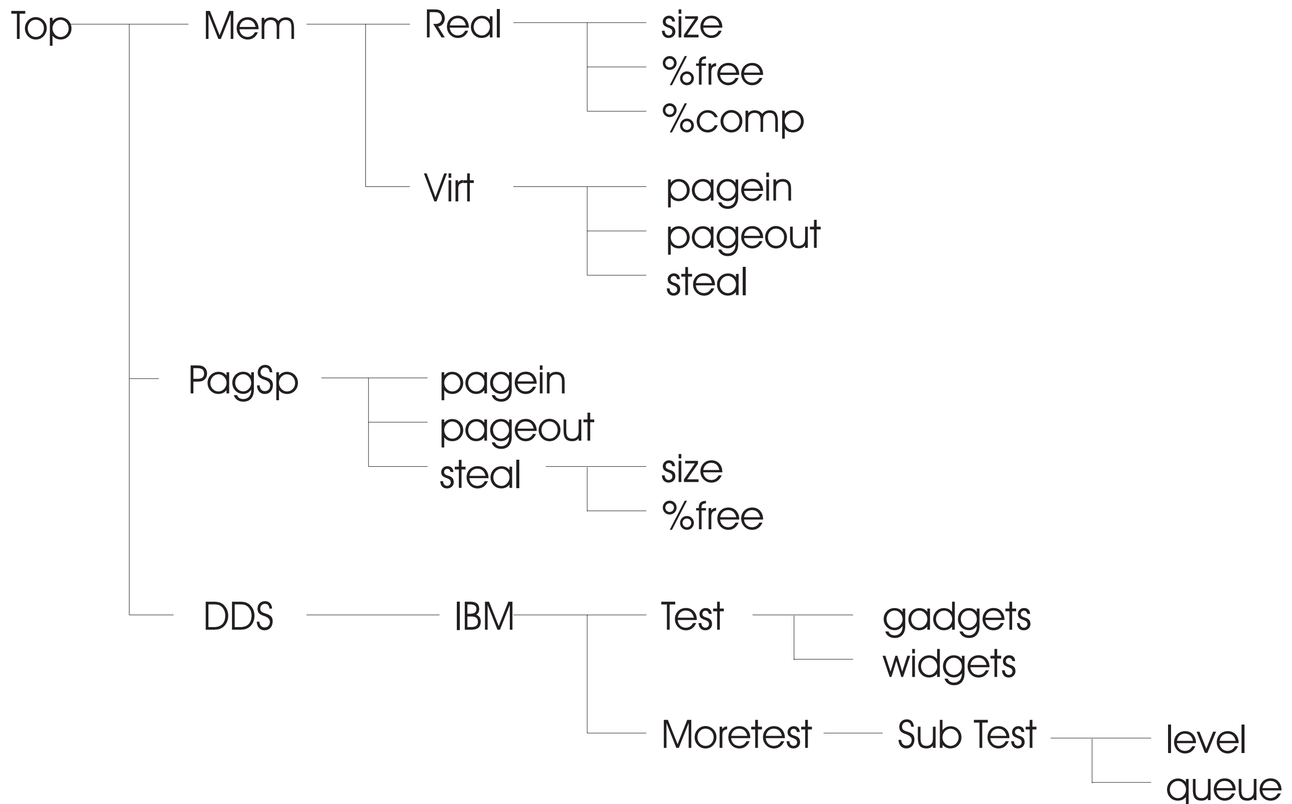



*Figure 13. Extended Set of Statistics. This figure shows a set of subcontexts have been added to the DDS context.*

In the preceeding figure, two contexts have been added as subcontexts of a context called **DDS/IBM** and are named *Test* and *Moretest*. The first of these contexts has two statistics called *gadgets* and *widgets*. The second has no directly descendent statistics but has a subcontext called *SubTest*, which in turn has two statistics: *level* and *queue*.

By convention, DDS programs always add statistics below the context **DDS/vendor** where *vendor* is the name of the vendor or customer that develops the DDS program; not the name of the machine type that the programs run on. This convention is established to prevent name clashes between the DDS programs developed by different vendors. Statistics should only be added to subcontexts of the **DDS/vendor** contexts, never to the **DDS/vendor** context itself.

The hierarchy shown in Figure 13 could be displayed with the program **xmpeek**. This generates output as follows:

```
/birte/Mem/             Memory statistics
/birte/Mem/Real/           Physical memory statistics
/birte/Mem/Real/size           Size of physical memory (4K pages)
/birte/Mem/Real/%free           % memory which is free
/birte/Mem/Real/%comp           % memory allocated to computational segments
/birte/Mem/Virt/           Virtual memory management statistics
/birte/Mem/Virt/pagein           4K pages read by VMM
/birte/Mem/Virt/pageout          4K pages written by VMM
/birte/Mem/Virt/steal            Physical memory 4K frames stolen by VMM
/birte/PagSp/           Paging space statistics
/birte/PagSp/size            Total active paging space size (4K pages)
/birte/PagSp/free            Total free disk paging space (4K pages)
/birte/PagSp/hd6/            Statistics for paging space hd6
/birte/PagSp/hd6/size            Size of paging space (4K pages)
/birte/PagSp/hd6/%free           Free portion of this paging space (percent)
/birte/DDS/             Dynamic Data-Supplier Statistics
/birte/DDS/IBM/            IBM-defined Dynamic Data-Suppliers
```

```
/birte/DDS/IBM/Test/           Bogus Context Number 1
/birte/DDS/IBM/Test/gadgets          Fake counter value
/birte/DDS/IBM/Test/widgets          Another fake counter value
/birte/DDS/IBM/Moretest/       Bogus Context Number 2
/birte/DDS/IBM/Moretest/SubTest/          Bogus Context Number 3
/birte/DDS/IBM/Moretest/SubTest/level          Fake quantity value
/birte/DDS/IBM/Moretest/SubTest/queue          Another fake quantity value
```

## Dynamic Data Supplier for Permanent Extensions

For this first exercise, it is assumed that the added contexts and statistics are non-volatile and as such can be added as permanent statistics. This requires the use of only one subroutine and the following programming steps:

1. "Declare Data Structures to Describe Statistics."
2. "Declare Data Structures to Describe Contexts" on page 221.
3. "Declare Other Data Areas as Required" on page 222.
4. "Initialize the SPMI Interface" on page 223.
5. "Initialize Exception Handling" on page 223.
6. "Initialize Statistics Fields" on page 223.
7. "Create Main Loop" on page 224.

### Declare Data Structures to Describe Statistics

Statistics are described in a simple structure of type **structSpmiRawStat** ("Declaring a Statistic - the SpmiRawStat Structure" on page 213). For each of the contexts you define that has statistics, you must create a table of statistics. The definition of the statistics *gadgets* and *widgets* would look as follows:

```
static const struct SpmiRawStat PUStats[] = {
{ "gadgets", "Fake counter value", 0, 100, SiCounter,
   SiLong, 1, SZ_OFF(dat, a, SiULong)},
{ "widgets", "Another fake counter value", 0, 100,
SiCounter,
   SiLong, 2, SZ_OFF(dat, b, SiULong)},
};
```

The fields in the structure are the following:

- Short name of statistic, 32 bytes character data.
- Description of statistic, 64 bytes character data.
- Lower Range for plotting, numeric, less than upper range.
- Upper Range for plotting, numeric, higher than lower range.
- Symbolic Constant defining the way data values should be interpreted. Currently, only the following are defined:

**SiCounter** Value is incremented continuously. Usually, data-consumer programs show the delta (change) in the value between observations, divided by the elapsed time, representing a rate.

**SiQuantity** Value represents a level, such as memory used or available disk space.

- Symbolic Constant describing the format of data as it is delivered to the data consumers. The data format must be one of the types defined by the "enum" **DataType** ("SpmiStat Structure" on page 206 section) in the include file **/usr/include/sys/Spmidef.h**. Currently, only the types **SiLong** and **SiFloat** are valid. If any other type is specified, **SiFloat** is assumed.
- ASN.1 (Abstract Syntax Notation One) Number, or sequence number, used when statistics defined to the SPMI are exported to the SNMP (Simple Network Management Protocol). Each of the statistics belonging to a context must have a unique ASN.1 number.
- SZ_OFF Macro as defined in the include file **/usr/include/sys/Spmidef.h**. The macro takes three arguments as follows:

– Name of a structure containing the source data field for this statistics value.
– Name of the source data field for this statistics value in the structure named previously.
– Data format of the source data field.

Because you actually want to add two sets of statistics at two different places in the context hierarchy, you also need to declare the second set. The following code piece shows how that can be done:

```
static const struct SpmiRawStat FakeMemStats[] = {
{ "level", "Fake quantity value", 0, 100, SiQuantity,
   SiLong, 1, SZ_OFF(dat, c, SiULong)},
{ "queue", "Another fake quantity value", 0, 100,
SiQuantity,
   SiLong, 2, SZ_OFF(dat, d, SiULong)},
};
```

## Declare Data Structures to Describe Contexts

After you have the statistics declared, you need to link them to their parent contexts. This is also done by defining a table of data structures. You need a single table of structures holding all the contexts you want to define as permanent contexts. Each context requires one element of the type **cx_create** (see the “Declaring a Context - the cx_create Structure” on page 214section). To create the three contexts you wanted to add, declare the four contexts as shown in the following code segment:

```
static const cx_create cx_table[] = {
{"DDS/IBM", "IBM-defined Dynamic Data-Suppliers", 2, 0,
   NULL, 0, NULL, 0, NULL, SiNoInst},
{"DDS/IBM/Test", "Bogus Context Number 1", 220, 0,
   PUStats, STAT_L(PUStats), NULL, 0, NULL, SiNoInst},
{"DDS/IBM/Moretest", "Bogus Context Number 2", 221, 0,
   NULL, 0, NULL, 0, NULL, SiNoInst},
{"DDS/IBM/Moretest/SubTest", "Bogus Context Number 3", 222,
0,
   FakeMemStats, STAT_L(FakeMemStats), NULL, 0, NULL, SiNoInst}
};
```

The first context declared is the vendor context. Because DDS programs from a vendor may be started in any sequence, there is no guarantee that this context exists. All DDS programs, therefore, must attempt to add the vendor context. If the vendor context exists, the attempt is ignored; otherwise the context is added. Note that this is the only type of context that is handled this way. Attempts to add other contexts twice cause the API to return an error to your program.

Each context element must have the following fields:

- Full path name of context, 64 bytes character data.
- Description of context, 64 bytes character data.
- The ASN.1 number assigned to the context. Each context within a parent context must have a unique ASN.1 number. For vendor contexts, the ASN.1 number should be set equal to the SNMP Assigned Enterprise Number.
- This field provides compatibility with internal data tables. It must be specified as zero.
- Pointer to the table of statistics for this context or NULL if none are defined.
- Count of elements in the table of statistics for this context or zero if none are defined. If statistics are defined, use the macro **STAT_L** to get the number of table elements.
- This field provides compatibility with internal data tables. It must be specified as NULL.
- This field provides compatibility with internal data tables. It must be specified as zero.
- This field provides compatibility with internal data tables. It must be specified as NULL.
- A symbolic constant describing the type of instantiation available for this context. The include file /usr/include/sys/Spmidef.h defines three constants you can use. If the context you are defining never will be extended by addition of subcontexts dynamically, specify the constant **SiNoInst**; otherwise use the constant **SiContInst**. The last of the three instantiation types has no meaning for dynamic data-supplier

statistics. Certain restrictions apply when defining DDS contexts that are to be made available to the SNMP agent. Refer to “Limitations Induced by SMUX” on page 179 and “Instantiation Rules” on page 180 to learn about these restrictions.

## Declare Other Data Areas as Required

Your dynamic data-supplier program must define its own data areas as required. The structure and fields are defined as follows:

```
extern char SpmiErrmsg[];
struct dat
{
   u_long  a;
   u_long  b;
   u_long  c;
   u_long  d;
};
static int  CxCount = CX_L(cx_table); /* Count of contexts defined */
static SpmiShare *dataarea = NULL;    /* Shared memory pointer */
static struct dat *d = NULL;         /* Pointer to stats data area */
```

The first line declares an external variable used by the SPMI API to return an error text to the invoking program if an error occurs.

The next lines define the data structure where the raw statistics are calculated to present to System Performance Measurement Interface (SPMI) interface. The data area must hold all the data fields referenced by non-volatile statistics.

Next, define a counter that you will use the CX_L macro to initialize with the number of static contexts that you want to add. Finally, define a pointer that will eventually be initialized to point to the data area you share with the SPMI interface.

The SPMI interface and the DDS use shared memory to communicate between themselves. When programs share memory, conventions must be established and adhered to for the use of the shared memory areas. The shared memory used by the SPMI is divided into two main areas: the shared memory structured fields and the shared memory data area.

## Shared Memory Structured Fields

The shared memory area is created by subroutines and its control information and generated data structures should (with few exceptions) never be used or manipulated by the DDS program directly. You can see the control information as it is defined in the include file **/usr/include/sys/Spmidef.h** as the structure **SpmiShare** (“Dynamic Data Supplier (DDS) Program Structures” on page 211). The fields that must be used by the DDS program are:

**SiShGoAway**
: This flag may be set by any data-consumer program to indicate that some condition indicates that the DDS program is no longer needed or that its data is corrupted. Usually, when a DDS sees this flag, it should call the **SpmiExit** subroutine to free the shared memory it allocated and to unlink from the SPMI interface. Failure to call **SpmiExit** before exiting retains the allocated DDS shared memory which, in turn, renders it impossible to restart the DDS program until the shared memory is freed through the command **ipcrm**.

**SiShT**
: A time stamp which must be updated by the DDS program each time the shared data area is updated. Data-consumer programs, and the **xmservd** daemon in particular, checks this field to see when your DDS was last active. If more than 30 seconds elapse without the time stamp being updated, **xmservd** assumes your dynamic data-supplier has died, sets the **SiShGoAway** flag and releases its access of the shared memory area. (30 seconds is the default. A value from 15 to 600 seconds can be specified using the command line option **-m** for **xmservd**.

  **Note:** The time stamp is a structure with two integer elements. It is expected to be stored in Big-Endian notation. Hosts that use Little-Endian notation must convert the integer fields to Big-Endian notation before storing them.

**SiShArea** The address of the data area in the shared memory segment. Your DDS program must load a pointer with the contents of this field and use that pointer to access the shared memory data area.

## Shared Memory Data Area

The shared memory data area is where your DDS is supposed to place its statistics values as they are calculated. You can do your calculations directly in the area allocated in shared memory, or you can do the calculations in local data fields and then move the result to shared memory. The important thing is to be aware that the shared memory area is guaranteed to be large enough to contain the last of those fields in your data structure that are referenced in any one of the tables defining statistics, but no larger.

Thus, if the structure **dat** as defined in the code segment in section “Declare Other Data Areas as Required” on page 222 had additional data fields, those would not be available in shared memory because no declared statistics reference them. Attempts to access such fields would cause segmentation faults.

## Initialize the SPMI Interface

With all required declarations in place, you can register with the SPMI interface. This is done through a single subroutine called **SpmiDDsAddCx**. For the purpose of this example, the subroutine is invoked with the following statements:

```
dataarea = SpmiDdsInit(cx_table, CxCount, NULL,0,
"/etc/SpmiSupl1SHM");
if (!dataarea)
{
   printf("%s", SpmiErrmsg);
   exit(-1);
}
d = (struct dat *) &dataarea->SiShArea[0];
```

## Initialize Exception Handling

Because a DDS uses shared memory to talk to SPMI, it is important to make sure the shared memory area is released when your DDS program dies. The best way to make sure this happens is to catch the signals that indicate that your program dies. The same function used to process the signals can conveniently be used for typical program exit. This could be done as shown in this code piece:

```
void SpmiStopMe()
{
   dataarea = NULL;
   SpmiExit();
   exit(0);
}
signal(SIGTERM, SpmiStopMe);
signal(SIGHUP, SpmiStopMe);
signal(SIGINT, SpmiStopMe);
signal(SIGSEGV, SpmiStopMe);
```

The function **SpmiStopMe** makes sure the shared memory area is freed and then exits. The “signal” lines defining the signal handler should be placed around the place in the DDS program where the program registers with SPMI.

## Initialize Statistics Fields

In most cases, statistics values are a combination of the types **SiCounter** and **SiQuantity**. Data consumers usually are interested in delta values for the former, so the first thing to do is take the first reading and initialize the statistics fields in shared memory. That way, even the first delta values read by a data consumer are likely to be valid.

Updating data fields always requires updating the time stamp. The lines used to do this and to give the initial field values could follow the scheme that follows. In this example, the fields are updated directly in the shared memory data area.

```
gettimeofday(&dataarea->SiShT, NULL);
d->a = ... ;
d->b = ... ;
d->c = ... ;
d->d = ... ;
```

## Create Main Loop

The main loop is usually simple and is conveniently made as a `while` loop. Always include in your `while` loop a test for the **SiShGoAway** flag. Your program may have additional conditions added to terminate the program as required by the application. The following example main loop only tests for the flag:

```
while(!dataarea->SiShGoAway)
{
   usleep(499000);
   gettimeofday($dataarea->SiShT, NULL);
   d->a = ... ;
   d->b = ... ;
   d->c = ... ;
   d->d = ... ;
}
SpmiStopMe();
```

Although the main loop can be as simple as shown previously, such simplicity may cause the DDS program to update the values in the shared memory area more often than required. In situations where the DDS has defined values but no data-consumer program is using any of those, updating the data fields is entirely unnecessary.

Two fields let you add a little more finesse to your dynamic data-supplier program. Both fields are Shared Memory Structured Fields and can be accessed through the pointer returned by **SpmiDDsAddCx**. The fields are not updated by every data-consumer program; only by the **xmservd** daemon. Therefore, your DDS program must be able to cope with the situation that the two fields are not updated. The fields are:

**SiShInterval**
An integer that gives you the number of milliseconds between requests for data values from **xmservd**. Because different requestors of values may request with different intervals, this value reflects the smallest interval of those defined (i.e., the interval defined for the instrument that runs fastest).

**SiShSubscrib**
The number of data values currently being requested from this DDS program by **xmservd**.

Obviously, if **SiShSubscrib** is zero, nobody is requesting continuous supply of data values and you can reduce the update frequency in your DDS accordingly. It is recommended that you do not stop the updating of the data fields but that you do so with intervals of, say, five seconds.

If **SiShSubscrib** is nonzero, somebody is requesting continuous supply of data values, so adjust the update frequency to match the request frequency as given in **SiShInterval**.

A main loop that uses these principles could look as shown here:

```
while(!dataarea->SiShGoAway)
{
   if (datarea->SiShSubscrib)
      usleep(dataarea->SiShInterval * 1000);
   else
      sleep(5);
   gettimeofday(&dataarea->SiShT, NULL);
   d->a = ... ;
   d->b = ... ;
   d->c = ... ;
   d->d = ... ;
}
SpmiStopMe();
```

The **SiShSubscrib** field usually holds a count of all data-consumer programs written to the RSi API (see Chapter 19, “Remote Statistics Interface Programming Guide,” on page 245) that are currently subscribing to data values in the shared memory area. However, in order to allow a program that acts both as a data consumer and a dynamic data supplier, you can move the port number of the port assigned to the data consumer side of the program to the field **SiShPortNo**, which is another shared memory structured field. A data-consumer/dynamic data-supplier program could use a statement like the following to insert the port number:

```
dataarea->SiShPortNo = rsh->portno;
```

where **rsh** is the **RSiHandle** for the host. The field **portno** in the **RSiHandle** structure is updated by the **RSiOpen** subroutine.

When the port number is inserted in the shared memory area, the **xmservd** does not count subscriptions for data values in the shared memory area that originate at that port number on the local host.

## The Entire Program

The program shown previously in segments is combined into a working DDS program that follows. Source code for the program can be found in **/usr/samples/perfagent/server/SpmiSupl1.c**:

```
#include stdio.h
#include sys/signal.h
#include sys/types.h
#include netinet/in.h
#include sys/Spmidef.h

extern char  SpmiErrmsg[];
struct dat
{
   u_long      a;
   u_long      b;
   u_long      c;
   u_long      d;
};
static const struct SpmiRawStat PUStats[] = {
{ "gadgets", "Fake counter value", 0, 100, SiCounter,
   SiLong, 1, SZ_OFF(dat, a, SiULong)},
{ "widgets", "Another fake counter value", 0, 100,
SiCounter,
   SiLong, 2, SZ_OFF(dat, b, SiULong)},
};
static const struct SpmiRawStat FakeMemStats[] = {
{ "level", "Fake quantity value", 0, 100, SiQuantity,
   SiLong, 1, SZ_OFF(dat, c, SiULong)},
{ "queue", "Another fake quantity value", 0, 100,
SiQuantity,
   SiLong, 2, SZ_OFF(dat, d, SiULong)},
};
static const cx_create cx_table[] = {
{"DDS/IBM", "IBM-defined Dynamic Data-Suppliers", 2, 0,
   NULL, 0, NULL, 0, NULL, SiNoInst},
{"DDS/IBM/Test", "Bogus Context Number 1", 220, 0,
   PUStats, STAT_L(PUStats), NULL, 0, NULL, SiNoInst},
{"DDS/IBM/Moretest", "Bogus Context Number 2", 221, 0,
   NULL, 0, NULL, 0, NULL, SiNoInst},
{"DDS/IBM/Moretest/SubTest", "Bogus Context Number 3", 222,
0,
   FakeMemStats, STAT_L(FakeMemStats), NULL, 0, NULL, SiNoInst},
};
static int  CxCount = CX_L(cx_table); /* Count of contexts defined */
static SpmiShare *dataarea = NULL;    /* Shared memory pointer  */
static struct dat *d = NULL;          /* Pointer to stats data area */

void SpmiStopMe()
{
   dataarea = NULL;
```

```
   SpmiExit();
   exit(0);
}

void main()
{
   dataarea = SpmiDdsInit(cx_table, CxCount, NULL, 0,
"/etc/SpmiSul1SHM");
   if (!dataarea)
   {
      printf("%s", SpmiErrmsg);
      exit(-1);
   }

   d = (struct dat *)&dataarea->SiShArea[0];
   signal(SIGTERM, SpmiStopMe);
   signal(SIGHUP, SpmiStopMe);
   signal(SIGINT, SpmiStopMe);
   signal(SIGSEGV, SpmiStopMe);

   gettimeofday &dataarea >SiShT, NULL);
   d->a = 22;
   d->b = 42;
   d->c = 28;
   d->d = 62;


while(!dataarea->SiShT, NULL);
{
   usleep(499000);
   gettimeofday(&dataarea->SiShT, NULL);
   d->a += dataarea->SiShT.tv_sec & 0xff;
   d->b += dataarea->SiShT.tv_sec & 0xf;
   d->c += (dataarea->SiShT.tv_sec & 0x20f) & 0xffff;
   d->d += (dataarea->SiShT.tv_sec & 0x7f) & 0xffff;
 }
 SpmiStopMe();
}
```

## Dynamic Data Supplier for Volatile Extensions

A DDS program may allow contexts and statistics to be added and deleted on the fly. For example, assume your DDS is concerned with monitoring of the response times between pairs of network hosts. On even a small network, it would be quite excessive to define all possible host pairs and keep track of them all. At any point in time, however, a limited number of sessions are active, but this number changes as do the host pairs involved. If you wanted to reflect this volatility in the statistics you present, you would need the ability to add and delete statistics on the fly.

To illustrate the use of the two subroutines that allow you to add and delete contexts dynamically, expand the first sample program. Extend the context hierarchy shown in Figure 13 on page 219 to look like the hierarchy in the following figure.

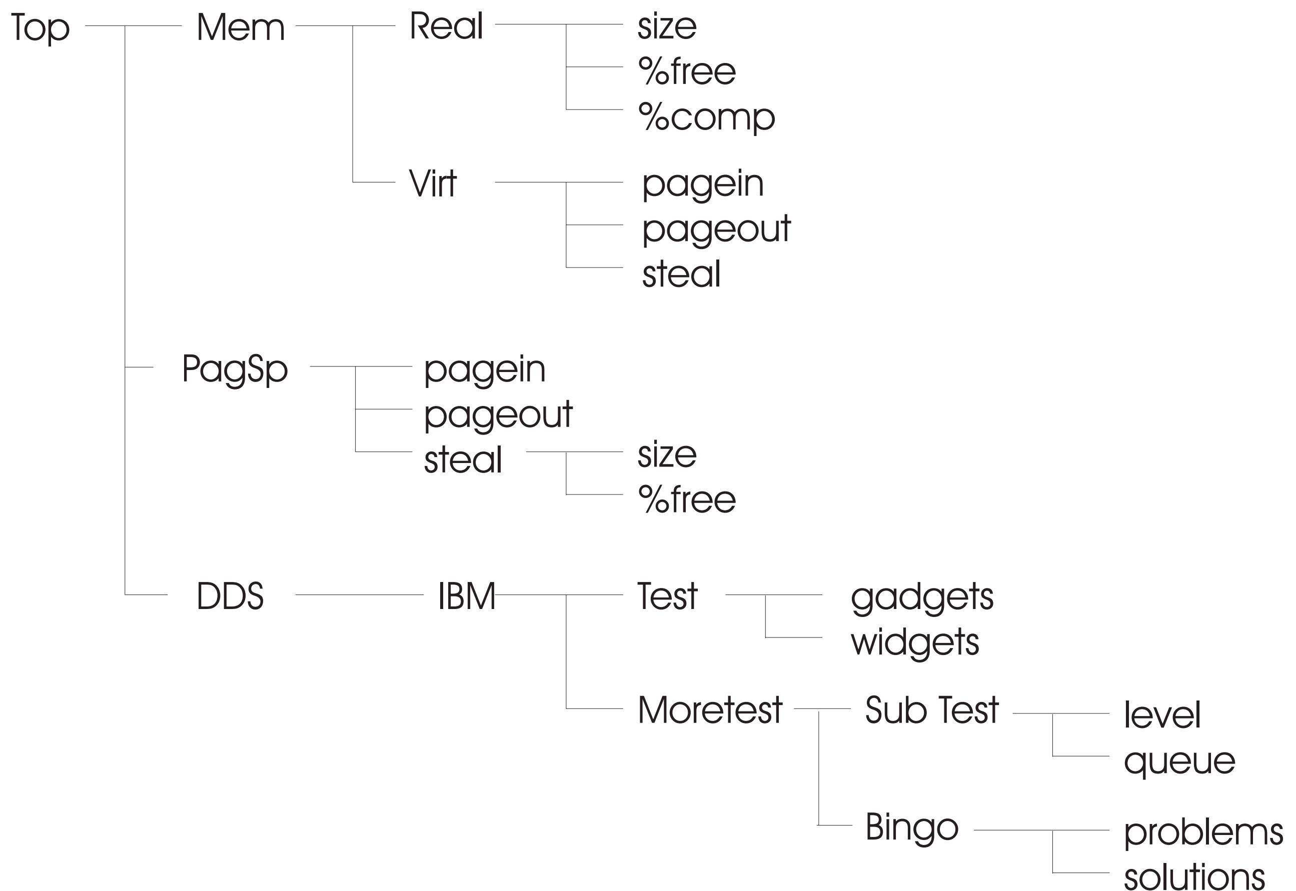



*Figure 14. Dynamic Extension of Statistics. This figure shows the hierarchy of a changing set of subcontexts.*

As you can see, there are plans to add a context called **Bingo** to the hierarchy with the previously added context **Moretest** as parent of the new context. The context that is added has two statistics values, namely **problems** and **solutions**. The plan is to allow the context to be added and deleted dynamically. For lack of a better trigger, let the time-of-day determine when to add and delete the context.

Rather than writing an entirely new program, take the previous example program and merely add the code lines required for the additional functionality using the following steps:

- “Declare Data Structures to Describe Dynamic Statistics.”
- “Declare Data Structures to Describe Dynamic Context” on page 228.
- “Declare Other Data Areas as Required” on page 228.
- “Modify Registration with the Spmi Interface” on page 228.
- “Modify Main Loop to Add and Delete Dynamic Context” on page 229.

## Declare Data Structures to Describe Dynamic Statistics

Statistics are defined almost the same way whether they are to be added permanently or dynamically. It is still true that all statistics for a context must be defined in one array. That one array may be referenced by more contexts, if appropriate; but most likely, it is not. The only real difference is that each set of statistics meant to be added dynamically must reference a separate data structure as source of its data fields. This is quite different from permanent statistics where all statistics source fields must reside in a common structure.

Obviously, there’s a reason for this. Static data values occur only once. They all reside in one contiguous area in shared memory. Dynamic data values, on the contrary, may exist in multiple instances and may

come and go. They are allocated dynamically in shared memory when they are required and when the values are deleted, their shared memory areas are returned to the free list.

The definition of statistics to add the **problems** and **solutions** values is shown as follows:

```
static CONST struct SpmiRawStat InstStats[] = {
{ "problems", "Fake counter value", 0, 100, SiCounter,
   SiLong, 1, SZ_OFF(inst, a, SiLong)},
{ "solutions", "Another fake counter value", 0, 100,
SiCounter,
   SiLong, 2, SZ_OFF(inst, b, SiLong)} };
```

Notice that this time you do not reference the structure **dat** used previously, but a different structure called **inst**, which is yet to be defined.

## Declare Data Structures to Describe Dynamic Context

In this example, you add only a single context. You could have added many more but for each context, which the DDS program may want to add, one element must be defined in a table of contexts. No context can be dynamically added unless it was defined in a table and passed to the **SpmiDDsAddCx** function when your DDS registered with the SPMI. The table has exactly the same format as the table of permanent contexts but must not be the same table. The following code segment shows how to define the single context you need to add:

**Note:** The pack name, description, and ASN.1 number of the context are used as placeholders, only. The real values to use are supplied on the **SpmiDDsAddCx** subroutine each time a context is called.

```
static CONST cx_create inst_table[] = {
{"DDS/IBM/Moretest/INST1", "Instantiable Context Number 1",
215, 0,
   InstStats, STAT_L(InstStats), NULL, 0, NULL, SiNoInst}
};
```

## Declare Other Data Areas as Required

You need only define the structure referenced by the declared statistics and a pointer to be used for accessing the allocated shared data area. For convenience, also define an integer to hold the number of dynamic contexts:

```
struct inst
{
   float  a;
   u_long  b;
};

int   InstCount = CX_L(inst_table); /* Count of contexts defined */
struct inst *pt1 = NULL;           /* Pointer to stats data area */
```

## Modify Registration with the Spmi Interface

Registration with the SPMI is almost unchanged. All you need is to tell the subroutine where the dynamic context table is and how many elements it has. This is shown in the following code segment:

```
dataarea = SpmiDdsInit(cx_table, CxCount, inst_table, InstCount,
"/etc/SpmiSupl_hook");
if (!dataarea)
{
   fprintf(stderr, "%s\n", SpmiErrmsg);
   exit(-1);
}
d = (struct dat *)&dataarea->SiShArea[0];
```

## Modify Main Loop to Add and Delete Dynamic Context

Finally, the following code segment shows the modified main loop. The loop has been extended with three pieces of code. The first one uses a **SpmiDDsAddCx** subroutine to add the context. The second uses **SpmiDdsDelCx** to delete the context again, and the third updates the values in the shared data area whenever the context and its statistics are active:

```
while(!dataarea->SiShGoAway)
{
   if (dataarea->SiShSubscrib)
      usleep(dataarea->SiShInterval * 1000);
   else
      sleep(5);
   gettimeofday(&dataarea>SiShT, NULL);
   d->a = ... ;
   d->b = ... ;
   d->c = ... ;
   d->d = ... ;
   if (((dataarea->SiShT.tv_sec % 59) == 0) && (!pt1))
   {
      if (!(pt1 = (struct inst *)SpmiDdsAddCx(0,
"DDS/IBM/Moretest/Bingo",
           "Dynamically added", 1)))
         fprintf(stderr, "Add failed: \"%s\"\n",
SpmiErrmsg);
   }
   if (((dataarea->SiShT.tv_sec % 120) == 0) && (pt1))
   {
      if (i = SpmiDdsDelCx((char *)pt1))
         fprintf(stderr, "Delete failed: \"%s\"\n",
SpmiErrmsg);
      else
         pt1 = NULL;
   }
   if (pt1)
   {
      pt1->a = ... ;
      pt1->b = ... ;
   }
}
SpmiStopMe();
```

The supplied sample program **SpmiSupl.c** does what has just been explained. It also adds yet another context dynamically. You may want to play with that program before writing your own.

## Recognizing Volatile Extensions

When your dynamic data-supplier program adds or deletes volatile extensions, this is indicated to SPMI through fields in the shared memory area. Neither the **xmservd** daemon nor other local data-consumer programs become aware of your changes until some event prompts them to look in the shared memory area.

This approach keeps the updating of the context structure to a minimum. The changes are only implemented if requested. The following is a list of events that causes **xmservd** or other local data-consumer programs to check the shared memory area for changes to volatile extensions. The **RSi** calls represent requests received by **xmservd** over the network interface; the **SPMI** calls would be issued by **xmservd** because of incoming requests or by other local data-consumer programs as required:

- Whenever the **RSiPathGetCx** or the **SpmiPathGetCx** subroutine is used on any of the contexts defined by your DDS (that is, whenever a program attempts to find a context pointer from a value path name). This function is usually required for any traversal of the context hierarchy.
- Whenever the **RSiFirstCx** or the "SpmiFirstCx Subroutine" on page 340 subroutine is used on any of the contexts defined by your DDS (that is, whenever a program starts traversing the subcontexts of a context in your DDS).

- Whenever the **RSiFirstStat** or the **SpmiFirstStat** subroutine is used on any of the contexts defined by your DDS (that is, whenever a program starts traversing the statistics of a context in your DDS).
- Whenever the **RSiInstantiate** or the **SpmiInstantiate** subroutine is used on any of the contexts defined by your DDS (that is, whenever a program explicitly asks for instantiation of any of the contexts defined by your dynamic data-supplier program).

## Example of an SPMI Data User Program

The following example program accesses the SPMI data:

```
/*  The following statistics are added by the SpmiPathAddSetStat
 *  subroutine to form a set of statistics:
 *     CPU/cpu0/kern
 *     CPU/cpu0/idle
 *     Mem/Real/%free
 *     PagSp/%free
 *     Proc/runque
 *     Proc/swpque
 *  These statistics are then retrieved every 2 seconds and their
 *  value is displayed to the user.
 */

#include sys/types.h
#include sys/errno.h
#include signal.h
#include stdio.h
#include sys/Spmidef.h

#define TIME_DELAY 2            /* time between samplings */

extern char   SpmiErrmsg[];     /* Spmi Error message array */
extern int    SpmiErrno;        /* Spmi Error indicator */

struct SpmiStatSet *statset;    /* statistics set */

/*====================== must_exit() ==========================*/
/* This subroutine is called when the program is ready to exit.
 * It frees any statsets that were defined and exits the
 * interface.
 */
/*=============================================================*/

void must_exit()
{
    /* free statsets */
    if (statset)
    if (SpmiFreeStatSet(statset))
       if (SpmiErrno)
       printf("%s", SpmiErrmsg);

    /* exit SPMI */
    SpmiExit();
    if (SpmiErrno)
       printf("%s", SpmiErrmsg);

    exit(0);
}

/*======================== getstats() =========================*/
/* getstats() traverses the set of statistics and outputs the
 * statistics values.
 */
/*=============================================================*/

void getstats()
{
    int                 counter=20;    /* every 20 lines output
                                        * the header
                                        */

    struct SpmiStatVals   *statval1;
    float                 spmivalue;
```

```
    /* loop until a stop signal is received. */
    while (1)
    {
       if(counter == 20)
       {
          /* output header info */
          /* The statistics are displayed in reverse order of how
           * they were entered into the set of statistics.
           */
          printf("\nCPU/cpu0   CPU/cpu0  Mem/Real   PagSp     ");
          printf("Proc       Proc\n");
          printf("    kern       idle    %%free     %%free    ");
          printf("runque     swpque\n");
          printf("===========================================");
          printf("===============\n");
          counter=0;
       }

       /* retrieve set of statistics */
       if (SpmiGetStatSet(statset, TRUE) != 0)
       {
          printf("SpmiGetStatSet failed.\n");
          if (SpmiErrno)
             printf("%s", SpmiErrmsg);
          must_exit();
       }

       /* retrieve first statistic */
       statval1 = SpmiFirstVals(statset);
       if (statval1 == NULL)
       {
          printf("SpmiFirstVals Failed\n");
          if (SpmiErrno)
             printf("%s", SpmiErrmsg);
          must_exit();
       }

       /* traverse the set of statistics */
       while (statval1 != NULL)
       {
          /* value to be displayed */
          Spmivalue = SpmiGetValue(statset, statval1);
          if (spmivalue < 0.0)
          {
             printf("SpmiGetValue Failed\n");
             if (SpmiErrno)
                printf("%s", SpmiErrmsg);
             must_exit();
          }
          printf("  %6.2f   ",spmivalue);

          statval1 = SpmiNextVals(statset, statval1);
       }  /* end while (statval1) */
       printf("\n");

       counter++;
       sleep(TIME_DELAY);
    }

    return;
}
/*======================== addstats() ========================*/
/* addstats() adds statistics to the statistics set. */
/* addstats() also takes advantage of the different ways a
 * statistic may be added to the set.
 */
/*============================================================*/
```

```
void addstats()
{
    SpmiCxHdl   cxhdl, parenthdl;

    /* initialize the statistics set */
    statset = SpmiCreateStatSet();
    if (statset == NULL)
    {
       printf("SpmiCreateStatSet Failed\n");
       if (SpmiErrno)
          printf("%s", SpmiErrmsg);
       must_exit();
    }

    /* Pass SpmiPathGetCx the fully qualified path name of the
     * context
     */
    if (!(cxhdl = SpmiPathGetCx("Proc", NULL)))
    {
       printf("SpmiPathGetCx failed for Proc context.\n");
       if (SpmiErrno)
          printf("%s", SpmiErrmsg);
       must_exit();
    }

    /* Pass SpmiPathAddSetStat the name of the statistic */
    /* & the handle of the parent */
    if (!SpmiPathAddSetStat(statset,"swpque", cxhdl))
    {
       printf("SpmiPathAddSetStat failed for Proc/swpque
          statistic.\n");
       if (SpmiErrno)
          printf("%s", SpmiErrmsg);
       must_exit();
    }

    if (!SpmiPathAddSetStat(statset,"runque", cxhdl))
    {
       printf("SpmiPathAddSetStat failed for Proc/runque
          statistic.\n");
       if (SpmiErrno)
          printf("%s", SpmiErrmsg);
       must_exit();
    }

    /* Pass SpmiPathAddSetStat the fully qualified name of the
     * statistic
     */
    if (!SpmiPathAddSetStat(statset,"PagSp/%free", NULL))
    {
       printf("SpmiPathAddSetStat failed for PagSp/%%free
          statistic.\n");
       if (SpmiErrno)
          printf("%s", SpmiErrmsg);
       must_exit();
    }

    if (!(parenthdl = SpmiPathGetCx("Mem", NULL)))
    {
       printf("SpmiPathGetCx failed for Mem context.\n");
       if (SpmiErrno)
          printf("%s", SpmiErrmsg);
       must_exit();
    }

    /* Pass SpmiPathGetCx the name of the context */
    /* & the handle of the parent context */
    if (!(cxhdl = SpmiPathGetCx("Real", parenthdl)))
    {
       printf("SpmiPathGetCx failed for Mem/Real context.\n");
```

```
      if (SpmiErrmsg)
         printf("%s", SpmiErrmsg);
      must_exit();
   }
   if (!SpmiPathAddSetStat(statset,"%free", cxhdl))
   {
      printf("SpmiPathAddSetStat failed for Mem/Real/%%free
         statistic.\n");
      if (SpmiErrno)
         printf("%s", SpmiErrmsg);
      must_exit();
   }
   /* Pass SpmiPathGetCx the fully qualified path name of the
    * context
    */
   if (!(cxhdl = SpmiPathGetCx("CPU/cpu0", NULL)))
   {
      printf("SpmiPathGetCx failed for CPU/cpu0 context.\n");
      if (SpmiErrno)
         printf("%s", SpmiErrmsg);
      must_exit();
   }
   if (!SpmiPathAddSetStat(statset,"idle", cxhdl))
   {
      printf("SpmiPathAddSetStat failed for CPU/cpu0/idle
         statistic.\n");
      if (SpmiErrno)
         printf("%s", SpmiErrmsg);
      must_exit();
   }
   if (!SpmiPathAddSetStat(statset,"kern", cxhdl))
   {
      printf("SpmiPathAddSetStat failed for CPU/cpu0/kern
         statistic.\n");
      if (SpmiErrno)
         printf("%s", SpmiErrmsg);
      must_exit();
   }
   return;
}
/*==============================================================*/
main(int argc, char **argv)
{
   int   spmierr=0;
   /* Initialize SPMI */
   if ((spmierr = SpmiInit(15)) != 0)
   {
      printf("Unable to initialize SPMI interface\n");
      if (SpmiErrno)
         printf("%s", SpmiErrmsg);
      exit(-98);
   }
   /* set up interrupt signals */
   signal(SIGINT,must_exit);
   signal(SIGTERM,must_exit);
   signal(SIGSEGV,must_exit);
   signal(SIGQUIT,must_exit);

   /* Go to statistics routines. */
   addstats();
   getstats();
```

```
    /* Exit SPMI */
    must_exit();
}
```

## Example of an SPMI Data Traversal Program

The following SPMI example program traverses a data hierarchy:

```
#include sys/types.h
#include sys/errno.h
#include stdio.h
#include sys/Spmidef.h

extern char         SpmiErrmsg[];       /* Error Msg array */


char                stats[256];         /* statistic name info */
char                cxt[256];           /* context name info */
char                subcxt[256];        /* text holder */
char                *blanks=" ";        /* blanks for text */
char                *blank2=" ";
int                 instantiable=0;
/*====================== findstats() ========================*/
/* findstats is a function that traverses recursively down a
 * context link.  When the end of the context link is found,
 * findstats traverses down the statistics links and writes the
 * statistic name to stdout.  findstats is originally passed the
 * context handle for the TOP context.
 */
/*==========================================================*/
void findstats(SpmiCxHdl cxhdl)
{
   struct SpmiCxLink   *cxlink;
   struct SpmiStatLink *statlink;
   struct SpmiCx       *spmicx, *spmicxparent;
   struct SpmiStat     *spmistat;
   char                *statname;
   char                num_string[30];
   int                 cxtlen1, cxtlen2, descplace;

  /* Get first context */
   if (cxlink = SpmiFirstCx(cxhdl))
   {

      while (cxlink)
      {

         /* output context name */
         spmicx = SpmiGetCx(cxlink->context);

         /* if the subcxt is a child of another context */
         if (strcmp(subcxt,NULL))
         {
            strcat(subcxt,"/");
            strcat(subcxt,spmicx->name);
         }
         else
            strcpy(subcxt,spmicx->name);

         cxtlen1 = strlen(spmicx->name);
         cxtlen2 = strlen(subcxt);

        /* determine if the context's parent is instantiable */
         /* because you don't want to have to print stats twice */
         spmicxparent = SpmiGetCx(spmicx->parent);
         if ( spmicxparent->inst_freq == SiContInst )
```

```
{
   instantiable++;
}
else
   instantiable = 0;

/* only want to print out the stats for any contexts */
/* whose parents aren't instantiable.  If the parent */
/* is instantiable then you only want to print out   */
/* the stats for the first instance of that parent. */
if (instantiable <= 1)
{
   strcpy(cxt,subcxt);
   if (!instantiable)
   {
      if (cxtlen1 == cxtlen2)
         descplace = 30 - cxtlen2;
      else
         descplace = 35 - cxtlen2;
      strncat(cxt,blanks,descplace);
      strcat(cxt,spmicx->description);
   }

  fprintf(stdout,"%s\n",cxt);

   /* Traverse the stats list for the context */
   if (statlink = SpmiFirstStat(cxlink->context))
   {
      while (statlink)
      {
         spmistat = SpmiGetStat(statlink->stat);
         statname = SpmiStatGetPath(cxlink->context,
         statlink->stat, 10);

         /* output statistic name */
         strcpy(stats,statname);
         descplace = strlen(stats);
         descplace = 40 - descplace;
         strncat(stats,blanks,descplace);
         strcat(stats,spmistat->description);
         fprintf(stdout, "%s\n",stats);

         /* output stat info */
         strcpy(stats,"Data Type(");
         if (spmistat->data_type == SiLong)
            strcat(stats,"Long) ");
         else
            strcat(stats,"Float)");
         strcat(stats,"  Value Type(");
         if (spmistat->value_type == SiCounter)
            strcat(stats,"Counter)");
         else
            strcat(stats,"Quantity)");
         fprintf(stdout, "%s%s\n",blank2,stats);
         /* output max/min info */
         sprintf(num_string,"min = %ld   max =
         %ld",spmistat->min,spmistat->max);
         strcpy(stats,num_string);
         fprintf(stdout, "%s%s\n",blank2,stats);

         /* Go to next statistic */
         statlink = SpmiNextStat(statlink);
      }  /* end while(statlink) */
   }  /* end if (statlink) */
}  /* end if (instantiable) */
else
{
```

```
            /* print out stat name info for stats with */
            /* instantiable parents */
            strcpy(cxt,spmicxparent->name);
            strcat(cxt,"/");
            strcat(cxt,spmicx->name);
            strcat(cxt,"/.....");
            fprintf(stdout,"%s\n",cxt);
         }           cxtlen1 = strlen(spmicx->name)+6;

/* recursive call to function */
         /* this gets the next context link */
         findstats(cxlink->context);

         if (cxtlen2 == cxtlen1)
           strcpy(subcxt,NULL);
         else
            subcxt[cxtlen2-cxtlen1-1] = NULL;

         /* Go to next context */
         cxlink = SpmiNextCx(cxlink);
      }  /* end while(cxlink) */
   }  /* end if (cxlink) */
   return;
}

/*========================= lststats() ========================*/
/* lststats gets the TOP context handle.  This handle is then
 * passed to the findstats routine
 */
/*============================================================*/

void lststats()
{
   SpmiCxHdl      cxhdl;

     if ((cxhdl = SpmiPathGetCx(NULL, NULL)) == NULL)
     {
      fprintf(stderr, "SpmiPathGetCx failed.\n");
      if (strlen(SpmiErrmsg))
         fprintf(stderr, "%s", SpmiErrmsg);
      return;
     }

     /* routine to traverse the context links */
     findstats(cxhdl);

   return;
}
/*============================  main()  =======================*/
main(int argc, char **argv)
{
   int    spmierr=0;
   /* Initialize SPMI interface */
   if ((spmierr = SpmiInit(15)) != 0)
   {
      fprintf(stderr, "Unable to initialize SPMI interface\n");
      fprintf(stderr, "%s", SpmiErrmsg);
      exit(-98);
   }
   /* Traversal routine. */
   lststats();

   /* Exit SPMI Interface */
   SpmiExit();
```

```
   if (strlen(SpmiErrmsg))
      fprintf(stderr, "%s", SpmiErrmsg);
   exit(0);
}
```

## Example of an SPMI Dynamic Data-Supplier Program

The following SPMI example program expands the data hierarchy:

```
/*
==============================================================
   This module is a sample data supplier module for the Spmi
   interface.  It is provided only as an example and has no
   practical use whatsoever.
   ==============================================================
*/

#include stdio.h
#include sys/signal.h
#include sys/types.h
#include netinet/in.h
#include sys/Spmidef.h

#ifdef _AIX
#define CONST  const
#else
#define CONST
#endif

extern char    SpmiErrmsg[];

/*
   The data area where statistics are passed to the Spmi
   interface must be defined as a structure (not typedef'ed).
   The structure can reside in local memory and be copied to
   shared memory whenever new statistics values are calculated --
   or it can be updated directly in shared memory as this module
   does.

   Please note that shared memory is NOT reserved for the entire
   structure size unless the last field in the structure is
   referenced in the table of statistics referring to the
   structure.  For example, the structure "dat" defines 6 4-byte
   long integers but only the first four are referenced in the
   statistics table.  The shared memory area reserved is thus
   4x4 = 16 bytes.  Attempts to reference the two last elements
   in shared memory will cause a segmentation fault or destroy
   other data areas.

   The following structure is used as data area definition for the
   sample program.
*/
struct dat
{
   u_long      a;
   u_long      b;
   u_long      c;
   u_long      d;
   u_long      e;
   u_long      f;
};/*
   The following two tables of type (struct SpmiRawStat) define
   two sets of statistics.  You must define one table for each
   set of statistics you have.  A set of statistics is defines as
   all statistics that have identical path names, except for the
   name of the statistic itself.  For example,
```

```
      DDS/IBM/Bingo/players/losers and
      DDS/IBM/Bingo/players/winners

   belong to the same set of statistics, while

      DDS/IBM/Bingo/players/losers and
      DDS/IBM/Blackjack/players/losers

   belong to two different sets.  When the (struct SpmiRawStat)
   entry is defined in a dynamic data supplier program, the last
   field in the structure need not be specified or must be
   specified as NULL.  Other fields must be filled in as follows:

      Field    Format      Contents
      -----    --------    -------------------------------------
        1      char[32]    Short name of statistic
        2      char[64]    Description of statistic
        3      numeric     Lower range for plotting
        4      numeric     Upper range for plotting
        5      ValType     See Spmidef.h
        6      DataType    Data format to deliver to consumer
                           (see Spmidef.h)
        7      numeric     The ASN.1 number assigned to the
                           statistic
        8.1    structure   Name of defined statistics structure
        8.2    fieldname   Name of data field in statistics
                           structure
        8.3    DataType    Data format of field in statistics
                           structure

*/
static CONST struct SpmiRawStat PUStats[] = {
   {"gadgets", "Fake counter value", 0, 100, SiCounter,
      SiLong, 1, SZ_OFF(dat, a, SiULong), NULL},
   {"widgets", "Another fake counter value", 0, 100,
SiCounter,
      SiLong, 2, SZ_OFF(dat, b, SiULong), NULL},
};
static CONST struct SpmiRawStat FakeMemStats[] = {
   {"level", "Fake quantity value", 0, 100, SiQuantity,
      SiLong, 1, SZ_OFF(dat, c, SiULong), NULL},
   {"queue", "Another fake quantity value", 0, 100,
SiQuantity,
      SiLong, 2, SZ_OFF(dat, d, SiULong), NULL},
};/*
   The following table defines the tree structure of contexts as
   defined by this module.  Each context is defined by one table
   entry.  The fields in the contexts are:

      Field    Format      Contents
      -----   --------    -------------------------------------
        1     char[64]    Full path name of the context to create
        2     char[64]    Description of context
        3     numeric     ASN.1 number to be assigned to the
                          context
        4     pointer     Pointer to statistics table, NULL if
                          none
        5     numeric     Count of elements in above statistics
                          table
                          Use STAT_L to find number of elements.
        7     pointer     Must be specified as NULL
        8     numeric     Must be specified as 0 (zero)
        9     pointer     Must be specified as NULL
       10     SiInstFreq  See Spmidef.h

*/
 static CONST cx_create cx_table[] = {
```

```
   {"DDS/IBM", "IBM-defined Dynamic Data Suppliers", 2, 0,
      NULL, 0, NULL, 0, NULL, SiNoInst},
   {"DDS/IBM/sample1", "Bogus Context Number 1", 191, 0,
      PUStats, STAT_L(PUStats), NULL, 0, NULL, SiNoInst},
   {"DDS/IBM/sample2", "Bogus Context Number 2", 192, 0,
      NULL, 0, NULL, 0, NULL, SiNoInst},
   {"DDS/IBM/sample1/SubContext", "Bogus Context Number 3",
      193, 0, FakeMemStats, STAT_L(FakeMemStats), NULL, 0, NULL,
      SiNoInst},
};

static int       CxCount = CX_L(cx_table);  /* Count of
                                             * context defined
                                             */
static SpmiShare *dataarea = NULL;          /* Shared memory
                                             * pointer
                                             */
static struct    dat *d = NULL;             /* Pointer to stats
                                             * data area
                                             */
/*
   This subroutine will make sure the shared memory allocated by
   this module is released before the module exits.  The
   subroutine is called whenever the module is about to exit.
*/
void SpmiStopMe(sig)
int     sig;
{
   if (sig)
      printf("SiSupl killed by signal %d\n", sig);
   else
      printf("SiSupl exiting (%s)\n", SpmidmiErrmsg);
   SpmiExit();
   dataarea = NULL;
   exit(0);
}/*
   SiSupl module main function - starts by checking that the
   program is executed with (required) root credentials.
*/

void main()
{
   if (geteuid())
   {
      fprintf(stderr,"Root authority req'd\n");
      exit(0);
   }
   /*
      Call the SpmiDdsInit subroutine to allocate shared memory
      and contact the SPMI interface.  If this succeeds, the
      subroutine will return with the address of the shared
      memory.

      The first two arguments are the table of static contexts
      to create and the count of elements in that table.  The
      next two arguments are the table of instantiable contexts
      and the count of elements in that table.  The latter two can
      be NULL and zero if your data supplier module does not
      require contexts to be added or deleted on the fly.

      The last argument is the name of a file, which either (1)
      must exist and be writeable by the user executing this
      code, or (2) can be created by the user.  It is used to
      generate a key for the shared memory area and must, of
      course, be UNIQUE between data supplier modules.
   */
#ifdef _AIX
```

```
   dataarea = SpmiDdsInit(cx_table, CxCount, NULL, 0,
   "/etc/SiSuplSHM");
#else
   dataarea = SpmiDdsInit(cx_table, CxCount, NULL, 0,
   "/etc/SiSuplSHM", 8092);
#endif
   if (!dataarea)
   {
      printf("%s", SpmiErrmsg);
      exit(-1);
   }
   /*
      The field SiShArea in shared memory has the address of the
      area where you are supposed to deliver your statistics.  In
      this sample module, simply set a pointer to point at
      this data area.
   */
   d = (struct dat *)&dataarea->SiShArea[0];
   /*
      You've got the shared memory and SPMI registered your presence.
      Now make sure you free the shared memory area if you get
      killed.
   */   signal(SIGTERM, SpmiStopMe);
   signal(SIGHUP, SpmiStopMe);
   signal(SIGINT, SpmiStopMe);
   signal(SIGSEGV, SpmiStopMe);   /*
      It is usual and recommended that you initialize the data
      area with values that make sense, even before anybody uses
      the data. This includes setting the time stamp at the time
      you do it.
   */
   gettimeofday(&dataarea->SiShT, NULL);
   d->a = 22;
   d->b = 42;
   d->c = 28;
   d->d = 62;

   /*
      The module now runs as long as the flag SiShGoAway is
      false.  Data consumer programs may set this flag if
      abnormal conditions are detected.  For each iteration, the
      module produces whatever statistics it can supply.  In this
      module everything is bogus, and the loop is a sleep loop.
      In a real data supplier module, you could use any timer
      function to drive you, or you could depend on some external
      function waking you up.

      For each time the module updates the data area, the time
      must be set as shown in the gettimeofday() call.
   */
   while(!dataarea->SiShGoAway)
   {
      #ifdef _AIX
         usleep(499000);
      #else
         sleep(1);
      #endif
         gettimeofday(&dataarea->SiShT, NULL);
         d->a += dataarea->SiShT.tv_sec & 0xff;
         d->b += dataarea->SiShT.tv_sec & 0xf;
         d->c += (dataarea->SiShT.tv_sec & 0x20f) & 0xffff;
         d->d += (dataarea->SiShT.tv_sec & 0x7f) & 0xffff;
   }
   SpmiStopMe(0);
}
```

## SPMI Interface Subroutines

The SPMI subroutines are organized according to function.

The following functional lists of SPMI subroutines are provided:

- Initialize, Terminate, and Instantiate subroutines
- Data Hierarchy Traversal subroutines
- StatSet Maintenance subroutines
- HotSet Maintenance subroutines
- Data Access subroutines
- Expand or Reduce the Data Hierarchy subroutines.

### Initialize, Terminate, and Instantiate Subroutines

The following subroutines prepare a system for SPMI processing, free memory after SPMI processing, or create an instance of a system resource or object:

**SpmiGetCx**
Initializes the SPMI.

**SpmiExit**
Releases allocated memory and disconnects from the SPMI library.

**SpmiInstantiate**
Explicitly instantiates the subcontexts of an instantiable context.

### Data Hierarchy Traversal Subroutines

The following subroutines navigate through an SPMI data hierarchy:

**SpmiPathGetCx**
Returns a handle to use when referencing a context.

**SpmiFirstCx**
Locates the first subcontext of a context.

**SpmiNextCx**
Locates the next subcontext of a context.

**SpmiFirstStat**
Locates the first statistic belonging to a context.

**SpmiNextStat**
Locates the next statistic belonging to a context.

**SpmiStatGetPath**
Returns the full path name of a statistic.

**SpmiGetCx**
Returns a pointer to the "SpmiCx Structure" on page 206 structure corresponding to a specified context handle.

**SpmiGetStat**
Returns a pointer to the "SpmiStat Structure" on page 206 structure corresponding to a specified statistic handle.

### StatSet Maintenance Subroutines

The followings SPMI subroutines create or remove a set of statistics, or add or delete statistics from the set:

**SpmiCreateStatSet**
Creates an empty set of statistics.

**SpmiPathAddSetStat**
Adds a statistics value to a set of statistics.

**SpmiFreeStatSet**
Erases a set of statistics.

**SpmiDelSetStat**
Removes a single statistic from a set of statistics.

### HotSet Maintenance Subroutines

The followings SPMI subroutines create or remove a hotset, or add or delete sets of peer statistics to or from the set:

**SpmiCreateHotSet** Creates an empty set of peer statistics (hotset).

**SpmiAddSetHot** Adds a set of peer statistics values to a hotset.

**SpmiFreeHotSet** Erases a hotset.

**SpmiDelSetHot** Removes a set of peer statistics from a hotset.

## Data Access Subroutines

The following subroutines access SPMI statistics:

**SpmiGetStatSet** Requests the SPMI to read the data values for all statistics belonging to a specified statset.

**SpmiFirstVals** Returns a pointer to the first "SpmiStatVals Structure" on page 209 belonging to a statset.

**SpmiNextVals** Returns a pointer to the next "SpmiStatVals Structure" on page 209 belonging to a statset.

**SpmiGetValue** Returns a decoded value based on the type of data value extracted from the data field of an ("SpmiStatVals Structure" on page 209.

**SpmiFirstVals** Returns a pointer to the next "SpmiStatVals Structure" on page 209 in a set of statistics. Combines calls to **SpmiFirstVals**, **SpmiNextVals**, and **SpmiGetValue** into a single call for more efficient startset processing.

**SpmiGetHotSet** Requests the SPMI to read the data values for all sets of peer statistics belonging to a specified hotset.

**SpmiFirstHot** Returns a pointer to the first set of peer statistics ("SpmiHotVals Structure" on page 210) belonging to a hotset.

**SpmiNextHot** Returns a pointer to the next set of peer statistics ("SpmiHotVals Structure" on page 210) belonging to a hotset.

**SpmiNextHotItem** Returns a pointer to the next set of peer statistics ("SpmiHotVals Structure" on page 210) belonging to a hotset and decodes the next data value from the **SpmiHotVals** structure. Used to walk all the returned "SpmiHotItems" on page 211 elements returned by **SpmiGetHotSet**.

## Expand or Reduce the Data Hierarchy Subroutines

The following subroutines are used by SPMI Dynamic Data Supplier (DDS) programs:

"SpmiDdsInit Subroutine" on page 335 Initializes a DDS program, establishes access to the common shared memory area, and connects the DDS shared memory area to the SPMI.

**SpmiDdsAddCx** Adds a volatile context from a DDS program.

**SpmiDdsDelCx** Deletes a volatile context previously added from the DDS program.

# List of SPMI Error Codes

All SPMI subroutines use constants to define error codes. The SPMI Error Code table lists the error descriptions.

| Symbolic Name | Number | Description |
|---|---|---|
| **SiSuccess** | 0 | Successful execution. |
| **SiInvalidCx** | 180 | The referenced context doesn't exist. |
| **SiNoShmPtr** | 181 | Unable to identify shared memory segment. |
| **SiShmemFailed** | 182 | Unable to access shared memory. |

| Symbolic Name | Number | Description |
| --- | --- | --- |
| **SiAssumedDead** | 183 | A data-supplier program seems to have terminated. |
| **SiInstBusy** | 184 | Can't instantiate; shared memory update in progress. |
| **SiNotInst** | 185 | Requested instantiation could not be performed. |
| **SiNotInit** | 186 | SPMI interface not initialized. |
| **SiBadArgument** | 187 | One or more arguments to a subroutine call is not valid. |
| **SiNotFound** | 188 | A requested data structure doesn't exist. |
| **SiNoValue** | 189 | No data value returned from the **SpmiGetValue** subroutine. |
| **SiInitFailed** | 190 | Initializing of the SPMI API failed. |
| **SiSmuxFailed** | 191 | Context could not be exported to the **snmpd**daemon through the SMUX protocol. |
| **SiLocked** | 192 | Some other process or thread has locked the common shared memory area and prevents the subroutine from executing its the requested function. Retry the subroutine call. |
| **SiDuplicate** | 193 | Attempt to add a context path that already exists. |
| **SiCallocFailed** | 194 | Memory allocation error. |
| **SiNoLicense** | 195 | No license to use is installed and valid. |
| **SiDeleted** | 196 | The requested context has been deleted. |
| **SiOtherErr** | 197 | Unspecified internal error. |

# Chapter 19. Remote Statistics Interface Programming Guide

This chapter provides information about the Remote Statistics Interface.

- Remote Statistics Interface API Overview
- Remote Statistics Interface List of Subroutines
- RSI Interface Concepts and Terms
- A Simple Data-Consumer Program
- Expanding the Data-Consumer Program
- Inviting Data Suppliers
- A Full-Screen, Character-based Monitor
- List of RSi Error Codes.

## Remote Statistics Interface API Overview

An application programming interface (API) is available for those who want to develop programs that access the statistics available from one or more **xmservd** daemons. The API is called the Remote Statistics Interface or the RSI Interface. This chapter describes how you use the RSI Interface API by walking you through a couple of sample programs. Those sample programs, and others, are provided in machine-readable form as well. The sample programs can be found in the **/usr/samples/perfmgr** directory.

Use the RSI Interface API to write programs that access one or more **xmservd** daemons. This allows you to develop programs that print, post-process, or otherwise manipulate the raw statistics provided by the **xmservd** daemons. Such programs are known as *Data-Consumer programs*.

*AIX 5L Version 5.3 Technical Reference: Communications Volume 2* must be installed to see the RSi subroutines.

### Makefiles

The include files are based upon a number of define directives being properly set. They are usually defined with the **-D** preprocessor flag.

- _AIX Tells the include files to generate code for AIX.
- _BSD Required for proper BSD compatibility.

A Makefile to build all the sample programs provided could look like the one shown in the following listing:

```
LIBS = -L./ -lbsd -lSpmi
CC = cc
CFLAGS = -D_BSD -D_AIX

all:: RsiCons RsiCons1 chmon

RsiCons: RsiCons.c
      $(CC) -o RsiCons RsiCons.c $(CFLAGS) $(LIBS)

RsiCons1: RsiCons1.c
      $(CC) -o RsiCons1 RsiCons1.c $(CFLAGS) $(LIBS)

chmon: chmon.c $
      $(CC) -o chmon chmon.c $(CFLAGS) $(LIBS) -lcurses
```

If the system on which you compile doesn't support ANSI function prototypes, add the following flag:

```
-D_NO_PROTO
```

© Copyright IBM Corp. 1994, 2004 

## Remote Statistics Interface List of Subroutines

As interesting and useful as it may be to watch the graphics display of statistics shown by **xmperf**, many other uses of the wealth of statistics are available through the **xmservd** daemons on all the hosts in a network. The Remote Statistics Interface API allows you to create data-consumer programs that can get full access to the statistics of any host's **xmservd** daemon.

The RSI interface consists of several groups of subroutines that are discussed in the following section.

### Initialization and Termination

**RSiInit** Allocates or changes the table of RSI handles.

**RSiOpen** Initializes the RSI interface for a remote host.

**RSiClose** Terminates the RSI interface for a remote host and releases all memory allocated.

**RSiInvite** Invites data suppliers on the network to identify themselves and returns a table of data-supplier host names.

### Instantiation and Traversal of Context Hierarchy

**RSiInstantiate** Creates (instantiates) all subcontexts of a context object.

**RSiPathGetCx** Searches the context hierarchy for a context that matches a context path name.

**RSiFirstCx** Returns the first subcontext of a context.

**RSiNextCx** Returns the next subcontext of a context.

**RSiFirstStat** Returns the first statistic of a context.

**RSiNextStat** Returns the next statistic of a context.

### Defining Sets of Statistics to Receive

**RSiAddSetHot** Adds a single set of peer statistics to a hotset.

**RSiCreateHotSet** Creates an empty hotset.

**RSiCreateStatSet** Creates an empty statset.

**RSiPathAddSetStat** Adds a single statistic to a statset.

**RSiDelSetHot** Deletes a single set of peer statistics from a hotset.

**RSiDelSetStat** Deletes a single statistic from a statset.

**RSiStatGetPath** Finds the full path name of a statistic identified by an **SpmiStatVals** pointer.

### Starting, Changing and Stopping Data Feeding

**RSiStartFeed** Tells **xmservd** to start sending data feeds for a statset.

**RSiStartHotFeed** Tells **xmservd** to start sending hot feeds for a hotset.

**RSiChangeFeed** Tells **xmservd** to change the time interval between sending data feeds for a statset.

**RSiChangeHotFeed** Tells **xmservd** to change the time interval between sending hot feeds for a hotset.

**RSiStopFeed** Tells **xmservd** to stop sending data feeds for a statset.

**RSiStopHotFeed** Tells **xmservd** to stop sending hot feeds for a hotset.

## Receiving and Decoding Data Feed Packets

**RSiGetHotItem**
Returns the peer context name and data value for the first (next) **"SpmiHotItems" on page 211** element by extraction from data feed packet.

**RSiMainLoop**
Allows an application to suspend execution and wait to get waked up when data feeds arrive.

**RSiGetValue**
Returns data value for a given **SpmiStatVals** pointer by extraction from data feed packet.

**RSiGetRawValue**
Returns a pointer to a valid **SpmiStatVals** structure for a given **SpmiStatVals** pointer by extraction from data feed packet.

# RSI Interface Concepts and Terms

Before you start using the RSI interface API you need to be aware of the format and use of the RSI interface data structures. This section explains the structures and also introduces you to the commonalities of the library functions and to some important design concepts. This section has the following subsections:

- "RSI Interface Data Structures."
- "The RSI Request-Response Interface" on page 249.
- "The RSI Network Driven Interface" on page 249.
- "Resynchronizing" on page 250.

## RSI Interface Data Structures

The RSI interface is based upon control blocks (data structures) that describe the current view of the statistics on a remote host and the state of the interaction between a data consumer program and the remote host's **xmservd** daemon. Data structures to know about are as follows:

### RSI handle

An RSI handle is a pointer to a data structure of type **RSiHandleStruct**. Prior to using any other RSI call, a data-consumer program must use the **RSiInit** subroutine to allocate a table of RSI handles. An RSI handle from the table is initialized when you open the logical connection to a host and that RSI handle must be specified as an argument on all subsequent subroutines to the same host. Only one of the internal fields of the RSI handle should be used by the data-consumer program, namely the pointer to received network packets, **pi**. Only in very special cases will you ever need to use this pointer, which is initialized by **RSiOpen** and must never be modified by a data-consumer program. If your program changes any field in the RSI handle structure, results are highly unpredictable. The RSI handle is defined in **/usr/include/sys/Rsi.h**.

### SpmiStatVals

A single data value is represented by a structure defined in **/usr/include/sys/Spmidef.h** as **struct SpmiStatVals**. Be aware that none of the fields defined in the structure must be modified by application programs. The two handles in the structure are symbolic references to contexts and statistics and should not be confused with pointers. The last three fields are updated whenever a **data_feed** packet is received. These fields are as follows:

**val**
The latest actual contents of the statistics data field.

**val_change**
The difference (delta value) between the latest actual contents of the statistics data field and the previous value observed.

**error**
An error code as defined by the enum **Error** in include file **/usr/include/sys/Spmidef.h**.

Notice that the two value fields are defined as **union Value**, which means that the actual data fields may be long or float, depending on flags in the corresponding **SpmiStat** structure. The **SpmiStat** structure cannot be accessed directly from the **StatVals** structure (the pointer is not valid, as previously mentioned).

Therefore, to determine the type of data in the **val** and **val_change** fields, you must have saved the **SpmiStat** structure as returned by the **RSiPathAddSetStat** subroutine. This is rather clumsy, so the **RSiGetValue** subroutine does everything for you and you do not need to keep track of **SpmiStat** structures.

The **SpmiStat** structure is used to describe a statistic. It is defined in **/usr/include/sys/Spmidef.h** as type **struct SpmiStat**. If you ever need information from this data structure (apart from information that can be returned by the **RSiStatGetPath** subroutine) be sure to save it as it is returned by the **RSiPathAddSetStat** subroutine.

The **RSiGetRawValue** subroutine provides another way of getting access to an **SpmiStat** structure but can only do so while a data feed packet is being processed.

**SpmiStatSet** The **xmservd** daemon accepts the definition of sets of statistics that are to be extracted simultaneously and sent to the data-consumer program in a single data packet. The structure that describes such a set of statistics is defined in **/usr/include/sys/Spmidef.h** as of type **struct SpmiStatSet**. As returned by the **RSiCreateStatSet**, the **SpmiStatSet** pointer should be treated as a handle whose only purpose is to identify the correct set of statistics to several other subroutines.

When returned in a data feed packet, the **SpmiStatSet** structure holds the actual time the data feed packet was created (according to the remote host's clock) and the elapsed time since the latest previous data feed packet for the same **SpmiStatSet** was created.

**SpmiHotSet ("SpmiHotSet Structure" on page 210)**Represents another set of access structures that allow an application program to define an alternative way of extracting and processing metrics. They are used to extract data values for the most or least active statistics for a group of peer contexts. For example, it can be used to define that the program wants to receive information about the two highest loaded disks, optionally subject to the load exceeding a specified threshold.

When the SPMI receives a read request for an **SpmiHotSet**, the SPMI reads the latest value for all the peer sets of statistics in the hotset in one operation. This action reduces the system overhead caused by access of kernel structures and other system areas, and ensures that all data values for the peer sets of statistics within a hotset are read at the same time. The hotset may consist of one or many sets of peer statistics.

**SpmiHotVals** One **SpmiHotVals** structure is created for each set of peer statistics selected for the hotset. When the SPMI executes a request from the application program to read the data values for a hotset, all **SpmiHotVals** structures in the set are updated. The RSi application program can then traverse the list of **SpmiHotVals** structures by using the **RSiGetHotItem** subroutine call.

The **SpmiHotVals** structure carries the data values from the SPMI to the application program. Its data carrying fields are:

**error**
: Returns a zero value if the SPMI's last attempt to read the data values for a set of peer statistics was successful. Otherwise, this field contains an error code as defined in the **sys/Spmidef.h** file.

**avail_resp**
: Used to return the number of peer statistic data values that meet the selection criteria (threshold). The field **max_responses** determines the maximum number of entries actually returned.

**count**
: Contains the number of elements returned in the array **items**. This number will be the number of data values that met the selection criteria (threshold), capped at **max_responses**.

**items**
The array used to return **count** elements. This array is defined in the **SpmiHotItems** data structure. Each element in the **"SpmiHotItems" on page 211** array has the following fields:

**name**
The name of the peer context for which the values are returned.

**val**
Returns the value of the counter or level field for the peer statistic. This field returns the statistic's value as maintained by the original supplier of the value. However, the **val** field is converted to an SPMI data format.

**val_change**
Returns the difference between the previous reading of the counter and the current reading when the statistic contains counter data. When this value is divided by the elapsed time returned in the **"SpmiHotSet Structure" on page 210**, an event rate-per-time-unit can be calculated.

## The RSI Request-Response Interface

The RSI interface API has two distinctly different ways of operation. This section describes the RSI request-response protocol that sends a single request to **xmservd** and waits for a response. A timeout occurs if no response has been received within a specified time limit in which case one single retry is attempted. If the retry also results in a timeout, that fact is communicated to the caller by placing the constant **RSiTimeout** in the external integer field **RSiErrno**. If any other error occurred, the external integer field has some other non-zero value.

If neither a communications error nor a timeout occurred, a packet is available in the receive buffer pointed to by the **pi** pointer in the RSI handle. The packet includes a status code that tells whether the subroutine was successful at the **xmservd** side. You need only be concerned with checking the status code in a packet if it matters what exactly it is because the constant **RSiBadStat** is placed in **RSiErrno** to indicate to your program that a bad status code was received.

You can use the indication of error or success as defined for each subroutine to determine if the subroutine succeeded or you can test the external integer **RSiErrno**. If this field is **RSiOkay** the subroutine succeeded; otherwise it did not. The error codes returned in **RSiErrno** are defined in the enum **RSiErrorType**.

All the library functions use the request-response interface, except for **RSiMainLoop** (which uses a network driven interface) and **RSiInit**, **RSiGetValue**, and **RSiGetRawValue** (that do not involve network traffic).

## The RSI Network Driven Interface

The **xmquery** protocol, which is described in detail in "The xmservd Interface", defines three types of data packets that are sent from the data supplier side (**xmservd**) without being solicited by a request packet. Those packet types are the **still_alive**, the **data_feed**, and the **except_rec** packets. The **still_alive** packets are handled internally in the RSI interface and require no programming in the data-consumer program.

The **data_feed** packets are received asynchronously with any packets produced by the request-response type subroutines. If a **data_feed** packet is received when processing a request-response function, control is passed to a callback function, which must be named when the RSI handle is initialized with the **RSiOpen** subroutine.

When the data-consumer program is not using the request-response functions, it still needs to be able to receive and process **data_feed** packets. This is done with the **RSiMainLoop** function, which invokes the callback function whenever a packet is received.

Actually, the data feed callback function is invoked for all packets received that cannot be identified as a response to the latest request sent, except if such packets are of type **i_am_back**, **still_alive**, or **except_rec**. Note that this means that responses to “request-response” packets that arrive after a timeout is sent to the callback function. It is the responsibility of your callback function to test for the packet type received.

The **except_rec** packets are received asynchronously with any packets produced by the request-response type subroutines. If an **except_rec** packet is received when processing a request-response function, control is passed to a callback function, which must be named when the RSI handle is initialized with the **RSiOpen** subroutine.

When the data-consumer program is not using the request-response functions, it still needs to be able to receive and process **except_rec** packets. This is done with the **RSiMainLoop** function which invokes the callback function whenever a packet is received.

Note that the API discards **except_rec** messages from a remote host unless a callback function to process the message type was specified on the **RSiOpen** subroutine call for that host.

## Resynchronizing

Network connections can go bad, hosts can go down, interfaces can be taken down and processes can die. In the case of the **xmservd** protocol, such situations usually result in one or more of the following:

**Missing packets**

Responses to outstanding requests are not received, which generate a timeout. That’s fairly easy to cope with because the data-consumer program has to handle other error return codes anyway. It also results in expected data feeds not being received. Your program may want to test for this happening. The proper way to handle this situation is to use the **RSiClose** function to release all memory related to the dead host and to free the RSI handle. After this is done, the data-consumer program may attempt another **RSiOpen** to the remote system or may simply exit.

**Resynchronizing requests**

Whenever an **xmservd** daemon hears from a given data-consumer program on a particular host for the first time, it responds with a packet of type **i_am_back**, effectively prompting the data-consumer program to resynchronize with the daemon. Also, when the daemon attempts to reconnect to data-consumer programs that it talked to when it was killed or died, it sends an **i_am_back** packet.

It is important that you understand how the **xmservd** daemon handles “first time contacted.” It is based upon tables internal to the daemon. Those tables identify all the data-consumers that the daemon knows about. Be aware that a data-consumer program is known by the host name of the host where it executes suffixed by the IP port number used to talk to the daemon. Each data-consumer program running is identified uniquely as are multiple running copies of the same data-consumer program.

Whenever a data-consumer program exits orderly, it alerts the daemon that it intends to exit and the daemon removes it from the internal tables. If, however, the data-consumer program decides to not request data feeds from the daemon for some time, the daemon detects that the data consumer has lost interest and removes the data consumer from its tables as described in “Life and Death of xmservd” on page 157. If the data-consumer program decides later that it wants to talk to the **xmservd** again, the daemon responds with an **i_am_back** packet.

The **i_am_back** packets are given special treatment by the RSI interface. Each time one is received, a resynchronizing callback function is invoked. This function must be defined on the **RSiOpen** subroutine.

Note that all data-consumer programs can expect to have this callback invoked once during execution of the **RSiOpen** subroutine because the remote **xmservd** does not know the data

consumer. This is usual and should not cause your program to panic. If the resynchronize callback is invoked twice during processing of the **RSiOpen** function, the open failed and can be retried, if appropriate.

## A Simple Data-Consumer Program

In this section, the use of the API is illustrated by creating a small data-consumer program to produce a continuous list of statistics from a host. The first version accesses only CPU-related statistics. It assumes you want to get your statistics from the local host unless you specify a host name on the command line. The program continues to display the statistics until it is killed. Source code for the sample program can be found in **/usr/samples/perfmgr/RsiCons1.c**.

### Initializing and Terminating the Program

The main function of the sample program uses the three subroutines as shown in the following code segment. The lines 12 through 15 use any command line argument to override the default host name obtained by the **uname** function. Then lines 17 through 28 initialize the RSI interface using the **RSiInit** and **RSiOpen** subroutines. The program exits if the initialization fails.

```
[01] extern char   RSiEMsg[];
[02] extern int   RSiErrno;
[03] char   host[64], apath[256], head1[24][10], head2[24][10];
[04] char   *nptr, **navn = &nptr, *dptr, **desc = &dptr;
[05] struct utsname  uname_struct;
[06] RSiHandle   rsh;
[07] struct SpmiStatVals *svp[24];
[08] int     lct = 99, tix = 0;
[09]
[10] main(int argc, char **argv)
[11] {
[12]     uname(&uname_struct);
[13]     strcpy(host, uname_struct.nodename);
[14]     if (argc > 1)
[15]        strcpy(host, argv[1]);
[16]
[17]     if (!(rsh = RSiInit(1)))
[18]     {
[19]        fprintf(stderr, "Unable to initialize RSI interface\n");
[20]        exit(98);
[21]     }
[22]     if (RSiOpen(rsh, 100, 2048, host, feeding, resync, NULL))
[23]     {
[24]        if (strlen(RSiEMsg))
[25]           fprintf(stderr, "%s", RSiEMsg);
[26]        fprintf(stderr, "Error contacting host\"%s\"\n", host);
[27]        exit(-99);
[28]     }
[29]     signal(SIGINT, must_exit);
[30]     signal(SIGTERM, must_exit);
[31]     signal(SIGSEGV, must_exit);
[32]     signal(SIGQUIT, must_exit);
[33]
[34]     strcpy(apath, "hosts/");
[35]     strcat(apath, host);
[36]     strcat(apath, "/");
[37]     lststats(apath);
[38]     RSiClose(rsh);
[39]     exit(0);
[40] }
```

The following lines (29-32) make sure that the program detects any attempt to kill or terminate it. If this happens, the function **must_exit** is invoked. This function has the sole purpose of making sure the association with the **xmservd** daemon is terminated. It does this as shown in the following piece of code:

```
void must_exit()
{
   RSiClose(rsh);
   exit(-9);
}
```

Finally, lines 34 through 36 prepare an initial value path name for the main processing loop of the data-consumer program. This is the way all value path names should be prepared. After doing this, the main processing loop in the internal function **lststats** is called. If this function returns, issue an **RSiClose** call and exit the program.

## Defining a Statset

Eventually, you want the sample of the data-consumer program to receive data feeds from the **xmservd** daemon. Thus, start preparing the **SpmiStatSet**, which defines the set of statistics with which you are interested. This is done with the **RSiCreateStatSet** subroutine.

```
[01] voidlststats(char *basepath)
[02] {
[03]    struct SpmiStatSet *ssp;
[04]    char     tmp[128];
[05]
[06]    if (!(ssp = RSiCreateStatSet(rsh)))
[07]    {
[08]       fprintf(stderr, "RsiCons1 can\'t create StatSet\n");
[09]       exit(62);
[10]    }
[11]
[12]    strcpy(tmp, basepath);
[13]    strcat(tmp, "CPU/cpu0");
[14]    if ((tix = addstat(tix, ssp, tmp, "cpu0")) == -1)
[15]    {
[16]       if (strlen(RSiEMsg))
[17]          fprintf(stderr, "%s", RSiEMsg);
[18]       exit(63);
[19]    }
[20]
[21]    RSiStartFeed(rsh, ssp, 1000);
[22]    while(TRUE)
[23]       RSiMainLoop(499);
[24] }
```

In the sample program, the **SpmiStatSet** is created in the local function **lststats** shown previously in lines 6 through 10.

Lines 12 through 19 invoke the local function addstat (“Adding Statistics to the Statset”), which finds all the CPU-related statistics in the context hierarchy and initializes the arrays to collect and print the information. The first two lines expand the value path name passed to the function by appending **CPU/cpu0**. The resulting string is the path name of the context where all CPU-related statistics for “cpu0” are held. The path name has the format **hosts/hostname/CPU/cpu0** without a terminating slash, which is what is expected by the subroutines that take a value path name as an argument. The function **addstat** is shown in the next section. It uses three of the traversal functions to access the CPU-related statistics.

## Adding Statistics to the Statset

```
[01] int addstat(int ix, struct SpmiStatSet *ssp, char *path, char *txt)
[02] {
[03]    cx_handle   *cxh;
[04]    int     i = ix;
[05]    char     tmp[128];
[06]    struct SpmiStatLink *statlink;
[07]
[08]    if (!(cxh = RSiPathGetCx(rsh, path)))
[09]    {
```

```
[10]       fprintf(stderr, "RSiPathGetCx can\'t access host %s (path %s)\n", host, path);
[11]       exit(61);
[12]    }
[13]
[14]    if ((statlink = RSiFirstStat(rsh, cxh, navn, desc)))
[15]    {
[16]       while (statlink)
[17]       {
[18]          if (i > 23)
[19]              break;
[20]          strcpy(head1[i], txt);
[21]          strcpy(head2[i], *navn);
[22]          strcpy(tmp, path);
[23]          strcat(tmp, "/");
[24]          strcat(tmp, *navn);
[25]          if (!(svp[i] = RSiPathAddSetStat(rsh, ssp, tmp)))
[26]             return(-1);
[27]          i++;
[28]          statlink = RSiNextStat(rsh, cxh, statlink, navn, desc);
[29]       }
[30]    }
[31]  return(i);
[32] }
```

The use of **RSiPathGetCx** by the sample program is shown in lines 8 through 12. Following that, in lines 14 through 30, two subroutines are used to get all the statistics values defined for the CPU context. This is done by using **RSiFirstStat** and **RSiNextStat**.

In lines 20-21, the short name of the context ("cpu0") and the short name of the statistic are saved in two arrays for use when printing the column headings. Lines 22-24 construct the full path name of the statistics value by concatenating the full context path name and the short name of the value. This is necessary to proceed with adding the value to the **SpmiStatSet** with the **RSiPathAddSetStat**. The value is added by lines 25 and 26.

<H3>Data-Consumer Initialization of Data Feeds

The only part of the main processing function in *the main section* yet to explain consists of lines 21 through 23. The first line simply tells the **xmservd** daemon to start feeding observations of statistics for an **SpmiStatSet** by issuing the **RSiStartFeed** subroutine call. The next two lines define an infinite loop that calls the function **RSiMainLoop** to check for incoming **data_feed** packets.

There are two more subroutines concerned with controlling the flow of data feeds from **xmservd**. Neither is used in the sample program. The subroutines are described in **RSiChangeFeed** and **RSiStopFeed**.

## Data-Consumer Decoding of Data Feeds

Whenever a **data_feed** is detected by the RSI interface, the data feed callback function defined in the **RSiOpen** subroutine is invoked, passing the RSI handle as an argument to the callback function. The sample program's callback function for data feeds is shown in the following example. Most of the lines in the function are concerned with printing headings after each 20 detail lines printed. This is in line numbers 9 through 19 and 26.

```
[01] void feeding(RSiHandle rsh, pack *p)
[02] {
[03]    int  i;
[04]    float f;
[05]    long  v;
[06]
[07]    if (p->type != data_feed)
[08]       return;
[09]    if (lct > 20)
[10]    {
[11]       printf("\n\n");
```

```
[12]       for (i = 0; i < tix; i++)
[13]          printf("%08s", head1[i]);
[14]       printf("\n");
[15]       for (i = 0; i < tix; i++)
[16]          printf("%08s", head2[i]);
[17]       printf("\n");
[18]       lct = 0;
[19]    }
[20]    for (i = 0; i < tix; i++)
[21]    {
[22]       v = RSiGetValue(rsh, svp[i]) * 10.0;
[23]       printf("%6d.%d", v/10, v%10);
[24]    }
[25]    printf("\n");
[26]    lct++;
[27] }
```

Actual processing of received statistics values is done by the lines 20-24. It involves the use of the library subroutine **RSiGetValue**. The following is an example of output from the sample program **RsiCons1**:

```
$ RsiCons1 umbra

    cpu0    cpu0    cpu0    cpu0    cpu0    cpu0    cpu0    cpu0
    user    kern    wait    idle  uticks  kticks  wticks  iticks
     0.0     0.0     0.0   100.0     0.0     0.0     0.0   100.0
     0.0     0.0     0.0   100.0     0.0     0.0     0.0    99.9
     0.2     3.1     0.0    96.5     0.2     3.2     0.0    96.6
     3.5     5.5     1.5    89.1     3.5     5.5     1.5    89.1
     5.8     3.4     0.0    90.8     5.8     3.4     0.0    90.8
     8.8     8.3     0.1    82.5     8.8     8.3     0.2    82.5
    67.5     2.4     3.0    27.0    67.5     2.3     2.9    26.9
    16.0     0.6     0.8    82.5    16.0     0.6     0.8    82.6
    67.5     5.0     0.0    27.3    67.5     5.0     0.0    27.3
    19.0     6.1     0.9    73.8    19.1     6.1     0.9    73.8
    22.5     0.8     1.6    75.0    22.5     0.8     1.6    74.9
    60.2     6.1     0.0    33.5    60.2     6.1     0.0    33.5
$
```

## An Alternative Way to Decode Data Feeds

If you need to know more about the data received in **data_feed** packets than what can be obtained using the**RSiGetValue** subroutine, you can use the library subroutine **RSiGetRawValue**.

## Expanding the Data-Consumer Program

A slightly more capable version of the sample program discussed in the previous sections is provided as /usr/samples/perfmgr/RsiCons.c. This program also lists the statistics with the short name **xfer** for all the disks found in the system where the daemon runs. To do so, the program uses some additional subroutines to traverse contexts as described in the following section.

## Traversing Contexts

The **adddisk** function in the following list shows how the **RSiFirstCx**, **RSiNextCx**, and the **RSiInstantiate** subroutines are combined with **RSiPathGetCx** to make sure all subcontexts are accessed. The sample program's internal function **addstat** is used to add the statistics of each subcontext to the **SpmiStatSet** in turn. A programmer who wanted to traverse all levels of subcontexts below a start context could easily create a recursive function to do this.

```
[01] int adddisk(int ix, struct SpmiStatSet *ssp, char *path)
[02] {
[03]    int      i = ix;
[04]    char     tmp[128];
[05]    cx_handle   *cxh;
[06]    struct SpmiStatLink *statlink;
[07]    struct SpmiCxLink *cxlink;
```

```
[08]
[09]    cxh = RSiPathGetCx(rsh, path);
[10]    if ((!cxh) || (!cxh->cxt))
[11]    {
[12]       if (strlen(RSiEMsg))
[13]          fprintf(stderr, "%s", RSiEMsg);
[14]       fprintf(stderr, "RSiPathGetCx can\'t access host %s (path %s)\n",
[15]       host, path);
[16]       exit(64);
[17]    }
[18]    if (rsh->pi->data.getcx.context.inst_freq == SiContInst)
[19]    {
[20]       if ((i = RSiInstantiate(rsh, cxh)))
[21]          return(-1);
[22]    }
[23]    if ((cxlink = RSiFirstCx(rsh, cxh, navn, desc)))
[24]    {
[25]       while (cxlink)
[26]       {
[27]         strcpy(tmp, path);
[28]         if (strlen(tmp))
[29]            strcat(tmp, "/");
[30]         if (*navn)
[31]            strcat(tmp, *navn);
[32]         if ((i = addstat(i, ssp, tmp, *navn)) == -1)
[33]         {
[34]            if (strlen(RSiEMsg))
[35]               fprintf(stderr, "%s", RSiEMsg);
[36]            exit(63);
[37]         }
[38]         cxlink = RSiNextCx(rsh, cxh, cxlink, navn, desc);
[39]       }
[40]    }
[41]    return(i);
[42] }
```

The output from the **RsiCons** program when **xmservd** runs on an AIX 4.1 host is shown in the following example.

```
$ RsiCons encee

     CPU     CPU     CPU     CPU  hdisk3  hdisk1  hdisk0     cd0
  uticks  kticks  wticks  iticks    xfer    xfer    xfer    xfer
    19.6    10.0     4.1    67.1     2.7     4.1     0.0     0.0
    10.9    15.3     8.2    65.3     0.0     8.2     0.0     0.0
     0.5     2.0     0.0    97.5     0.0     0.0     0.0     0.0
    10.5     4.0     0.0    85.5     0.0     0.0     0.0     0.0
    55.4     8.9     0.0    35.4     2.4     0.0     0.0     0.0
    19.0     5.5     0.0    75.5     0.0     0.0     0.0     0.0
     5.9     6.4     0.0    87.4     0.0     0.0     0.0     0.0
    10.5     7.0     0.0    82.5     0.0     0.0     0.0     0.0
     7.9     7.4     0.0    84.4     0.0     0.0     0.0     0.0
    88.5     8.5     3.0     0.0     9.5     4.5     0.0     0.0
    89.4     8.9     1.4     0.0     5.9     0.0     0.0     0.0
    92.5     5.5     2.0     0.0     9.0     8.5     0.0     0.0
    71.0     6.0    23.0     0.0    44.0    41.0     0.0     0.0
    37.9     2.4    58.9     0.4    67.9    61.4     0.0     0.0
    17.5     4.5     0.0    78.0     1.5     3.0     0.0     0.0
     0.5     1.5    10.0    88.0     7.5     1.5     0.0     0.0
$
```

## Inviting Data Suppliers

Sometimes you want to design programs that can present the end user with a list of potential data-supplier hosts rather than requiring the user to specify which host to monitor. The **RSiInvite** allows you to create such programs.

## Identifying Data Suppliers

The **RSiInvite** subroutine uses one or more of the following methods to obtain the Internet Protocol (IP) addresses to which an invitational **are_you_there** message can be sent. The last two methods depend on the presence of the **$HOME/Rsi.hosts** file. PTX also has alternative locations of the **Rsi.hosts** file. The three ways to invite data-supplier hosts are:

1. Unless instructed not to by the user, the broadcast address corresponding to each of the network interfaces of the local host is found. The invitational message is sent on each network interface using the corresponding broadcast address. Broadcasts are not attempted on the Localhost (loopback) interface or on point-to-point interfaces such as X.25 or SLIP (Serial Line Interface Protocol) connections.
2. If a list of Internet broadcast addresses is supplied in the file **$HOME/Rsi.hosts**, an invitational message is sent on each such broadcast address. Note that if you specify the broadcast address of a local interface, broadcasts are sent twice on those interfaces. You may want to use this as a feature in order to minimize the likelihood of the invitation being lost.
3. If a list of host names is supplied in the file **$HOME/Rsi.hosts**, the host IP address for each host in the list is looked up and a message is sent to each host. The look-up is done through a **gethostbyname()** call, so that whichever name service is active for the host where the data-consumer application runs is used to find the host address.

The file **$HOME/Rsi.hosts** has a simple layout. Only one keyword is recognized and only if placed in column one of a line. That keyword is:

```
nobroadcast
```

and means that the **are_you_there** message should not be broadcast using method 1 shown previously. This option is useful in situations where a large number of hosts are on the network and only a well-defined subset should be remotely monitored. To say that you don't want broadcasts but want direct contact to three hosts, your **$HOME/Rsi.hosts** file might look like this:

```
nobroadcast
birte.austin.ibm.com
gatea.almaden.ibm.com
umbra
```

This example shows that the hosts to monitor do not necessarily have to be in the same domain or on a local network. However, doing remote monitoring across a low-speed communications line is unlikely to be popular; neither with other users of that communications line nor with yourself.

Be aware that whenever you want to monitor remote hosts that are not on the same subnet as the data-consumer host, you must specify the broadcast address of the other subnets or all the host names of those hosts in the **$HOME/Rsi.hosts** file. The reason is that IP broadcasts do not propagate through IP routers or gateways.

The following example illustrates a situation where you want to do broadcasting on all local interfaces, want to broadcast on the subnet identified by the broadcast address `129.49.143.255`, and also want to invite the host called **umbra**. (The subnet mask corresponding to the broadcast address in this example is `255.255.240.0` and the range of addresses covered by the broadcast is `129.49.128.0 - 129.49.143.255`.)

```
129.49.143.255
```

If the **RSiInvite** subroutine detects that the name server is inoperational or has abnormally long response time, it returns the IP addresses of hosts rather than the host names. If the name server fails after the list of hosts is partly built, the same host may appear twice, once with its IP address and once with its host name.

The execution time of the **RSiInvite** subroutine depends primarily on the number of broadcast addresses you place in the **$HOME/Rsi.hosts** file. Each broadcast address increases the execution time with roughly 50 milliseconds plus the time required to process the responses. The minimum execution time of the

subroutine is roughly 1.5 seconds, during which time your application only gets control if callback functions are specified and if packets arrive that must be given to those callback functions.

## A Full-Screen, Character-based Monitor

Another sample program written to the data-consumer API is the program **chmon**. Source code to the program is in **/usr/samples/perfmgr/chmon.c**. The **chmon** program is also stored as an executable during the installation of the Manager component. This program uses the API and the curses programming interface to create a screen full of statistics as shown in the following example:

```
Data-Consumer API       Remote Monitor for host     Tue Apr 14 09:09:05
1992
CHMON Sample Program       ***   birte  ***        Interval:      5 seconds

% CPU                                              EVENTS/QUEUES  FILE/TTY
Kernel  13.3   |####                             | Pswitch  1295  Readch 24589
User    23.7   |#######                          | Syscall  6173  Writech 1646
Wait     6.5   |##                               | Reads     487  Rawin      0
Idle    56.1   |################                 | Writes    143  Ttyout   106
                                                   Forks       1  Igets   1763
PAGING counts   PAGING SPACE   REAL MEM  48MB      Execs       1  Namei    809
Faults    131   % Used  33.7   % Comp    68.0      Runqueue    1  Dirblk   174
Steals      0   % Free  66.2   % NonComp 15.0      Swapqueue   0  Reads     48
Reclaim     0   Size,MB   96   % Client   4.0                     Writes   143

PAGING page/s   DISK       Read  Write     %       NETWORK   Read  Write
Pgspin      0   ACTIVITY KB/sec  KB/sec  Busy      ACTIVITY KB/sec KB/sec
Pgspout     0   hdisk0      0.0    35.1  15.7      lo0         1.1    1.1
Pagein      0   hdisk1      0.0     0.0   0.0      tr0         1.1    0.0
Pageout    11   hdisk2      0.0     9.5   3.5
Sios       10   cd1         0.0     0.0   0.0

      Process wait     (514)   %cpu 63.2, PgSp: 0.0mb, uid:
      Process xlcentry (12657) %cpu 58.0, PgSp: 1.1mb, uid: birte
      Process make     (21868) %cpu 15.0, PgSp: 0.2mb, uid: birte
      Process make     (5998)  %cpu 15.0, PgSp: 0.1mb, uid: birte
```

The **chmon** command line is:

**chmon** **[**-*iseconds_interval*] **[**-*pno_of_processes*] **[***hostname***>]**

where:

*seconds_interval*
: Is the interval between observations. Must be specified in seconds. No blanks must be entered between the flag and the interval. Defaults to 5 seconds.

*no_of_processes*
: Is the number of “hot” processes to be shown. A process is considered “hotter” the more CPU it uses. No blanks must be entered between the flag and the count field. Defaults to 0 (no) processes.

*hostname*
: Is the host name of the host to be monitored. Default is the local host.

The sample program exits after 2,000 observations have been taken, or when you type the letter “q” in its window.

## List of RSi Error Codes

All RSI subroutines use constants to define error codes. The RSI Error Code table lists the error descriptions.

| Symbolic Name | Number | Description |
|---|---|---|
| **RSiTimeout** | 280 | A time-out occurred while waiting for a response to a request. |

| Symbolic Name | Number | Description |
|---|---|---|
| **RSiBusy** | 281 | An **RSiOpen** subroutine was issued, but another is already active. |
| **RSiSendErr** | 282 | An error occurred when the library attempted to send a UDP packet with the sendto() system call. |
| **RSiPollErr** | 283 | A system error occurred while issuing or processing a poll() or select() system call. |
| **RSiRecvErr** | 284 | A system error occurred while attempting to read an incoming UDP packet with the recvfrom() system call. |
| **RSiSizeErr** | 285 | A recvfrom() system call returned a UDP packet with incorrect length or incorrect source address. |
| **RSiResync** | 286 | While waiting for a response to an outgoing request, one of the following occurred and cause an error return to the calling program:<br>1. An error occurred while processing an exception packet.<br>2. An error occurred while processing an i_am_back packet.<br>3. An i_am_back packet was received in response to an output request other than are_you_there.<br>4. While waiting for a response to an outgoing request, some asynchronous function closed the handle for the remote host.<br><br>The code may also be set when a success return code is returned to the caller, in which case it shows that either an exception packet or an i_am_back packet was processed successfully while waiting for a response. |
| **RSiBadStat** | 287 | A bad status code was received in the data packet received. |
| **RSiBadArg** | 288 | An argument that is not valid was passed to an RSi subroutine. |
| **RSiBadHost** | 289 | A valid host address cannot be constructed from an IP address or the nameservice doesn't know the hostname. |
| **RSiDupHost** | 290 | An **RSiOpen** call was issued against a host but a connection is already open to a host with this IP address and a different hostname. |
| **RSiSockErr** | 291 | An error occurred while opening or communicating with a socket. |
| **RSiNoPort** | 292 | The RSi is unable to find the port number to use when inviting remote suppliers. The likely cause is that the xmquery entry is missing from the **/etc/services** file or the NIS (Yellow Pages) server. |
| **RSiNoMatch** | 293 | One of the following occurred:<br>1. The SpmiStatVals argument on the **RSiStatGetPath** call is not valid.<br>2. On an **RSiPathAddSetStat** call, the **SpmiStatSet** argument is not valid or the path name given in the last argument does not exist.<br>3. On an **RSiAddSetHot** call, the **SpmiHotSet** argument is not valid, the grand parent context doesn't exist or none of its subcontexts contain the specified statistic.<br>4. On an **RSiDelSetStat** call, the **SpmiStatSet** or the **SpmiStatVals** argument is not valid.<br>5. On an **RSiDelSetHot** call, the **SpmiHotSet** or the **SpmiHotVals** argument is not valid.<br>6. On an **RSiPathGetCx** call, the path name given does not exist.<br>7. On an **RSiGetValue** or **RSiGetRawValue** call, the **SpmiStatVals** argument is not valid<br>8. On an **RSiGetHotItem** call, the **SpmiHotSet** argument was not valid. |
| **RSiInstErr** | 294 | An error was returned when attempting to instantiate a remote context. |
| **RSiNoFeed** | 295 | When extracting a data value with the **RSiGetValue** call, the data value was marked as not valid by the remote data supplier. |

| Symbolic Name | Number | Description |
| --- | --- | --- |
| **RSiTooMany** | 296 | An attempt was made to add more values to a statset than the current buffer size permits. |
| **RSiNoMem** | 297 | Memory allocation error. |
| **RSiNotInit** | 298 | An RSi call was attempted before an **RSiInit** call was issued. |
| **RSiNoLicense** | 299 | License expired or no license found. |
| **RSiNotSupported** | 300 | The subroutine call requires a later protocol version that the one supported by the remote system's **xmservd**. |

# Chapter 20. Top Monitoring

Performance Toolbox Version 3.1 introduced *top monitoring* to simplify performance analysis of large server configurations. These systems typically contain a large number of CPUs, network adapters, and disk devices, which makes it difficult to visualize or represent system performance in a graphical manner. *Top monitoring*, also known as *hot monitoring*, focuses on elements consuming the most system resources and these elements are then sorted into lists referred to as *top-lists*. For example, the top ten processes consuming the most CPU resources would make up a *top-list*. System administrators and performance analysts can identify and diagnose issues of interest more quickly by focusing on the most constrained resources.

This chapter discusses the following topics:

- Top Monitoring Configuration
- Using the jtopas System-Monitoring Tool

**Note:** The Performance Aide for AIX trend agent, **xmtrend**, has been modified to support perpetual collection of top resource data. This agent is described in more detail in Chapter 11, “Analyzing Performance Trend Recordings with the jazizo Tool,” on page 139 and Chapter 14, “Recording Performance Data on Remote and Local Systems,” on page 167.

## Top Monitoring Configuration

The *top* framework records a defined number of performance metrics, by resource, at all times for all systems on which Performance Aide for AIX is installed. The framework consists of:

- User-centered distributed top resource client
- Always-on agent data collection and recording
- Tabular report summaries
- Near real-time response for active monitoring
- Playback function
- Common recording format allows support by existing trend analysis client (**jazizo**)

The jtopas client can retrieve this data for display and reporting. The **ptxtab** command-line utility can also output reports of data recordings. This utility is described in more detail in “ptxtab Command” on page 297.

No user configuration of the top agents’ recorded metric set is currently provided. The daemon records a predefined set of information for optimal performance on AIX systems. In addition, the daemon exports sampled data for near real-time retrieval by the jtopas client. The **xmtrend** agent that provides *top* data, operates independently of any other instance of a user-configured **xmtrend** or **xmservd** agent. That is, if a user intends to monitor or record other performance metrics, the user must manually configure the **xmtrend** or **xmservd** agents.

The top agent uses the **/usr/lpp/perfagent/jtopas.cf** configuration file. The only user-modifiable sections of this configuration file are:

- Retain Line
- Start-Stop Lines

These lines control recording retention and when a recording is active.

**Note:** Modification of other than these lines or removal of the **jtopas.cf** file disables the top agent.

© Copyright IBM Corp. 1994, 2004 

In addition to recording top resource data, the top agent also records a set of global system metrics that do not correspond to list data, for example, system memory size. The complete list of available global metrics is provided in the **jtopas.cf** configuration file.

Top data is recorded into the **/etc/perf/Top/** directory by default. Users requiring a separate recording file system for management purposes can create that file system and relink the **/etc/perf/Top** directory as desired. Top data recordings can also be viewed by the **jtopas** client or the **jazizo** trend analysis tool.

Upon installation, the top agent is added to the **/etc/inittab** file, so that it is enabled by default. Users who want to disable the top agent can comment out the **xmtrend** entry in the file.

Additional entries in the **/etc/inittab** file are the **tnameserv** program and the **feed** program. Both of these entries are required for the **jtopas** tool to register remote Java classes and communicate with the local and remote servers. These entries are added when Performance Toolbox for AIX is installed and can be disabled along with the top agent.

## Using the jtopas System-Monitoring Tool

The **jtopas** tool is a Java-based system-monitoring tool that provides a console to view a summary of the overall system, as well as separate consoles to focus on particular subsystems. Top instruments are featured in the**jtopas** tool for various resources such as processes and disks. The data streams available are Near Real-Time (NRT) and Playback (PB). PB data can be viewed from the local host or a remote host, as long as Performance Toolbox for AIX has been installed and configured.

The **jtopas** tool interface displays a set of tabs that represent the various consoles. The main console provides a view of several resources and subsystems and lends itself to providing an overall view of a computer system, while the other consoles focus more on particular areas of the system. The main console contains several top instruments. A top instrument is a monitoring window that displays a group of devices or processes. For instance, these top instruments can be sorted by the largest consumers of a system resource, such as memory, CPU, storage, or network adapters. Even though there might be thousands of processes, for example, only the top 10 or 20 are displayed by the **jtopas** tool.

Each of the other consoles is composed of one or more instruments. An instrument is similar to a window that can be resized, minimized, or moved. A divider bar is used to separate top-instrument information from global information about the system, and the bar can be moved or either side of the bar can be made to use the entire console display area.

At initialization, the **jtopas** tool displays all consoles with their instruments. If a user configuration file is found, the consoles are constructed based on that file. Otherwise, the default configuration is used. By default, the **jtopas** tool tries to establish a communication link with the local host to drive the consoles.

To run the **jtopas** tool, type:

```
jtopas
```

### Files Used by the jtopas Tool

The **jtopas** tool uses recording files and a configuration file, as follows:

**Recording Files**
Recording files contain metric values recorded by an instance of the **xmtrend** agent, acting as the top agent. This **xmtrend** agent is directed to record metric data specifically for top data. The **xmtrend** agent creates a recording file of top metric data as defined in the **jtopas.cf** configuration file. This recording file can be used by the **jtopas** tool to display historical system events, or by the **jazizo** trend analysis tool. Not all data and data rates are available to the **jtopas** tool during a playback. For top recordings and Near Real-Time data, the **xmtrend** daemon must be started with the **-T** option. The top recordings are placed in the **/etc/perf/Top/** directory.

**Configuration File**
The **jtopas** tool uses a default configuration file that determines the size, location, and metrics viewed for each instrument. If any instrument is changed, upon exit, users are asked if they want to save the current configuration. If Yes is selected, a configuration file is placed in the user's **HOME** directory and is named **.jtopas.cfg**. Users can return to using the default configuration by deleting the **/$HOME/.jtopas.cfg** file.

## Menus for the Jtopas Tool

This section describes the various menus associated with the **jtopas** tool. The following are the **jtopas** menus:

- File Menu
- Data Source Menu
- Reports Menu
- Host List
- Options Menu

### File Menu

The following option is contained in the File menu:

**Exit** Closes all windows and exits the **jtopas** tool. If the configuration has changed, the user is asked whether to save the new configuration.

### Data Source Menu

The following options are contained in the Data Source menu:

**Near Real-Time Data**
Changes the data stream to near real-time data. Near real-time data is gathered from a machine in real time and then made available to the **jtopas** tool. The refresh rate, which can be changed in the **jtopas** tool, defines how often data is requested and displayed.

**PlayBack Data**
Changes the data stream to PlayBack data. The PlayBack control panel is displayed when users select this option. The **jtopas** tool continues to display data at the refresh rate. The data is gathered from the local or a remote machine. Recorded data is saved on a server by the **xmtrend** agent at 1-minute intervals. Although the refresh rate updates the console at a given interval by default, the clock associated with the data increments at the 1-minute interval. For example, if the refresh rate is every 5 seconds and the recording file is recorded every minute, the data and clock on the PlayBack panel refreshes every 5 seconds by 1 minute.

### Reports Menu

The Reports menu provides a set of report formats. Each report summarizes the data in a tabular format that can be viewed and printed. The font and size of the data can be changed. Some reports might offer report options to change how the data is summarized and displayed.

### Host List

The Host List menu allows users to add or delete a host name from the host list.

### Options Menu

The following options are contained in the Options menu:

**Refresh Rate**
The **jtopas** tool cycles through at the refresh rate. The cycle includes requesting the data and updating the console. The refresh rate can be changed by either clicking the refresh rate/status button or selecting the menu option. The user can enter values of whole seconds. The **jtopas** tool uses the default refresh rate. The greater the refresh rate value, the less load the **jtopas** tool consumes on the CPU. If the **jtopas** tool is unable to complete an operation within the cycle time,

the status button turns yellow and an appropriate message is displayed. If data cycles are consistently missed, the refresh rate should be adjusted to increase the time between updates.

**Message Filter**
: The message filter option allows users to filter out and display messages based on a specific priority.

The following are priorities for messages, each priority having a color associated with it:

- Priority 1 Red - Critical message, such as losing a host connection
- Priority 2 Yellow - Important message, such as losing a data cycle
- Priority 3 Black - Informational messages

The text of each message displayed is color-coded and is preceded by the priority and the timestamp.

## Info Section for the jtopas Tool

The info section provides status information and allows users to select the host from which to gather the data. The following are the data fields:

**Host Name**
: By default, the local host name is displayed. Host names can be added, deleted, or selected.

To add a new host, select **Host List** from the menu bar and then select **Add Host**. The new host is immediately contacted for a connection and is added to the host list pull-down. If the host list is modified in any way, upon exit, the user is asked whether to save the new configuration. If **OK** is selected, the new host list is saved in the **$HOME/.jtopas.cfg** file and made available the next time the same user starts the **jtopas** tool.

To delete a host, select **Host List** from the menu bar and then select **Delete Host**. The old host is still selected until a new host is selected.

To select a new host from the host list, open the list and select the host name.

**Message Section**
: The **jtopas** tool generates informational messages. These messages are assigned a priority to classify them by importance and to allow users to hide messages of a particular priority for easier viewing. As stated in the **Message Filter** section of the Options menu, the following priorities are assigned to messages: **P1**, **P2**, or **P3**. The highest in importance is **P1**, as it is used for critical messages. Messages can be filtered by selecting **Message Filter** under the Options menu.

**Status/Refresh Rate Button**
: The status button reflects the status of data acquisition per the selected refresh rate. The refresh rate defines how often the console data is updated. The value is in seconds. The refresh rate can be changed by selecting the button or selecting **Refresh Rate** under the Options menu. If data is not retrieved and updated within the refresh cycle, the button turns yellow and the button label changes to **No Update**. If the data connection is lost, the button turns red and the button label displays **No Data**. Appropriate messages are also added to the message section.

**Current Time**
: This field reflects the current day and time.

## Consoles of the jtopas Tool

The various consoles are displayed under tabs on the interface. The initial console is the main console. By selecting a different tab, the corresponding console is activated and displayed. Each console contains one or more instruments. Each instrument is displayed as a window that can be minimized, maximized, moved, and resized. If there are multiple columns with headers, the columns can be reorganized and resized. Some instruments implement a scroll bar to view additional data.

Top instruments monitor a group of common metrics ordered by a particular column metric. For example, CPUs are by default ordered highest to lowest by largest consumer of kernel CPU used. This default can be changed to largest consumer of user CPU by clicking the **User** header label. Even if there are 64 CPUs, only a subset is displayed.

## PlayBack Panel for the jtopas Tool

When the PlayBack data source is selected, the PlayBack panel is displayed. The panel allows a user to control the playback. Closing the PlayBack panel returns the user to the NRT data source. Playbacks begin in a paused state. To begin displaying the playback, click **Play**. The PlayBack panel contains the following information:

**Host Name**
: The initial playback host is the host that was selected for the NRT data. This can be changed in the same manner as it is changed in the main console.

**Start / Stop**
: The available start and stop times of all recorded data on a particular host are displayed. By clicking **Change**, the start and stop date and times can be altered. The **Time Selection** panel displays dates and times of available recorded data. Select a date and indicate whether it is the start or stop date for the playback. Then select a start time and stop time. Click **OK** to use the dates and times selected.

**PlayBack Time**
: This time stamp represents the time stamp for the playback sample that is displayed.

**Sample Interval**
: Even though the recording frequency is in minutes, metric samples are taken at a much finer granularity. These samples are combined to determine the mean across the recording cycle. By default, sample updates to the **jtopas** tool in the playback mode are at the recording frequency. This is not the same as the refresh rate of the screen. The refresh rate represents how often the data in the **jtopas** console is refreshed. Having a refresh rate for the console, as well as a sample interval, allows the user to view a week's worth of data in hourly intervals and have the console refresh at a rate that is comfortable to view and analyze.

**PlayBack Controls**
: The following are the playback controls:

  **Rewind**
  : Plays the recording back in reverse. The sample interval value becomes negative, which indicates that the recording file is being traversed in reverse order and at the interval displayed. Each time **Rewind** is selected, the time interval increases. Clicking **Play** returns the playback to the default sample rate.

  **Play**
  : Displays the recording file.

  **Fast Forward**
  : Increases the time between data samples. The sample interval value increases, which indicates that the recording file is being traversed at greater intervals. Each time **Fast Forward** is selected, the time interval increases. Clicking **Play** returns the playback to the default or selected sample rate.

  **Pause**
  : Stops the playback but maintains the current playback time in the recording file.

  **Stop**
  : Stops the playback and resets the playback time to the beginning.

  **Step Forward**
  : Moves the playback forward one time interval and pauses.

  **Step Backward**
  : Moves the playback backward one time interval and pauses.

# Appendix A. Installing the Performance Toolbox for AIX

If you are installing Performance Toolbox for AIX, the distribution media contains two install images: one for the Agent component and one for the Manager component. If you are installing Performance Aide for AIX, the distribution media contains only the install image for the Agent component. In both cases, the install image for the Agent component contains two installable options: server and tools.

In AIX 4.3 of the operating system, the Performance Aide for AIX tools component (**perfagent.tools**) is shipped as an optionally installable component with the base operating system media. In AIX 4.3 and later releases of the operating system, the tools component must be installed before proceeding with the Agent or Manager installations.

## Prerequisites

Version 2 of Performance Toolbox for AIX runs only with AIX 4.1 or later releases of the operating system. Attempts to install Version 2 on other levels of the operating system may succeed, but the product will not run correctly.

## Ordering Information

These are the Performance Toolbox for AIX Version 2 feature codes:

**Performance Toolbox Network feature**
Performance Manager which allows monitoring of remote systems in a networked environment.

**Performance Toolbox Local feature**
Performance Manager which allows monitoring of the local system. If you have the Performance Toolbox Network feature listed above it contains this local feature.

**Performance Aide for 4.1**
Supplies data from AIX 4.1 of the operating system to either the Network or Local features listed above.

**Performance Aide for 4.2**
Supplies data from AIX 4.2 of the operating system to either the Network or Local features listed above.

**Performance Aide for 4.3**
Supplies data from AIX 4.3 of the operating system to either the Network or Local features listed above.

## Performance Aide for AIX (Performance Toolbox for AIX Agent)

The prerequisites for Version 2 for both Performance Aide for AIX and the Agent component of Performance Toolbox for AIX are the following:

- AIX Version 4 or later releases of the operating system
- **bos.net.tcp.client** at level 4.1 or higher (required for server option)
- **bos.sysmgt.trace** at level 4.1 or higher (required for tools option)

## Performance Toolbox for AIX Manager

The prerequisites for the Manager component of Performance Toolbox for AIX Version 2 are the following:

- AIX Version 4 or later releases of the operating system
- **X11.base.rte** at level 4.1 or higher
- **X11.base.lib** at level 4.1 or higher
- **X11.motif.lib** at level 4.1 or higher
- **perfagent.server** at level 2.1 or higher

© Copyright IBM Corp. 1994, 2004

- **perfmgr.common** at level 2.2 or higher

## Installation

Before you install Performance Toolbox for AIX on a system, you must decide whether you will be using that system to monitor other systems or if it is simply a system you want to monitor.

- If you plan to monitor only the local system, you need to install the complete Performance Toolbox Local feature (both the Manager and Agent components).
- If you plan to monitor local and remote systems, you need to install the complete Performance Toolbox Network feature (both the Manager and Agent components) on the system.
- If you plan to use local performance tools or monitor this system only from another system and not use this system to monitor other systems, you need to install either:
  - Performance Toolbox for AIX Agent component
  - Performance Aide for AIX (which provides Agent components for computers that are not RS/6000 computers).

### Performance Aide for AIX (Performance Toolbox for AIX Agent)

To install the Performance Aide for AIX, use SMIT or the following command (if installing from tape, otherwise substitute the proper installation device for `/dev/rmt0.1`):

```
installp -avgIX -d /dev/rmt0.1 perfagent
```

The Performance Aide server component has a prerequisite to the tools component. Because the tools component (**perfagent.tools**) resides on the base installation media, it must be installed before the server component (**perfagent.server**). The Agent installation can then be installed with the following command:

```
installp -avgIX -d /dev/rmt0.0 perfagent.server
```

### Performance Toolbox for AIX (Agent and Manager components)

To install the Performance Aide for AIX (both Agent and Manager components), use SMIT or the following command (if installing from tape, otherwise substitute the proper installation device for `/dev/rmt0.1`) as described here.

For the Performance Toolbox Network feature:

```
installp -avgIX -d /dev/rmt0.1 perfmgr.network
```

For the Performance Toolbox Local feature:

```
installp -avgIX -d /dev/rmt0.1 perfmgr.local
```

**Note:** If you use SMIT to install, make sure to request that prerequisite software be installed.

Before executing the Performance Toolbox Manager or Agent components, the Internet superserver must be updated. This is performed by executing the following command on each installed node:

```
refresh -s inetd
```

## Installing Performance Toolbox for AIX on Systems Other Than IBM RS/6000 Hosts

The actual installation procedure varies between systems. Specific installation instructions are provided in a readme file included with the software.

### Prerequisites

The prerequisites for installing an Agent component on a host that is not a RS/6000 host are the following:

- Performance Toolbox for AIX or Performance Aide for AIX is properly installed on a RS/6000 computer.

- TCP/IP network access (with root authority) to the host that is not a RS/6000 host.
- **/usr/lpp/perfagent/README.perfagent**

## Installation

After Performance Toolbox for AIX or Performance Aide for AIX is installed on a RS/6000, read the **/usr/lpp/perfagent/README.perfagent** file for further installation instructions.

# Appendix B. Performance Toolbox for AIX Files

Several of the programs in Performance Toolbox for AIX use files, either to customize the programs' behavior or as output or log files. The programs access their files according to an access scheme that allows a user of a host system to override the default information in the files without affecting the defaults available to other users. This appendix gives an overview of the files and their access scheme.

An access scheme is defined as:
- The directories where the the program attempts to locate the file.
- The sequence in which directories are searched.
- The possibility to override the above.

## Files used by xmperf and Other Data Consumers

The following files share a standard access scheme:

**xmperf.cf** The **xmperf** configuration file.

**xmperf.hlp** The **xmperf** simple help file.

**exmon.cf** The **exmon** configuration file.

**exmon.hlp** The **exmon** simple help file.

**azizo.hlp** The **azizo** simple help file.

**Rsi.hosts** The file that defines broadcast options to use.

**3dmon.cf** The **3dmon** configuration file.

**3dplay.hlp** The **3dplay** help file, available with Version 2.2 or later.

**filter.cf** The **filtd** configuration file

When a program needs one of these files, the file is first searched for in the user's home directory. If the file is found there, that file is used.

If the file doesn't exist in the user's home directory, the file is looked for in the directory **/etc/perf**. Because the **/etc** directory is always unique for all hosts, even when some hosts are diskless hosts, this allows for defining defaults on a per host basis. Again, if the file is found in this directory, that file is used.

The last place the file is looked for is in **/usr/lpp/perfmgr** (in case of the **filter.cf**, the last place the file is looked for is in **/usr/lpp/perfagent**). If the file cannot be located in any of the directories, the program will be missing important information and may terminate or provide reduced function.

The standard access scheme can be overridden by specifying a full file name on the command line when starting the program. This is the case for the files:

**xmperf.cf** The **xmperf** configuration file.

**3dmon.cf** The **3dmon** configuration file.

**filter.cf** The **filtd** configuration file.

The **xmperf** configuration file can be saved from the menus of **xmperf**. When this happens, the file is usually saved to the user's home directory. Only if the file name has been overridden on the **xmperf** command line, is the file saved to that same name. This gives users an easy way to modify the default configuration file.

© Copyright IBM Corp. 1994, 2004

The **filtd** program produces a log, which alternates between two file names that are identical, apart from one ending in 1 and the other in 2. The file names of the log files are as follows:

- **/etc/perf/filter.log1**
- **/etc/perf/filter.log2**

The **xmperf** and **azizo** programs use a log file to record any error or warning messages and state information. The file is written to **$HOME/xmperf.log** or **$HOME/azizo.log**, if possible, otherwise the output is directed to **stdout**. Each time **xmperf** and **azizo** execute, they overwrite any previous copy of their files.

## Files used by xmservd

The files **xmservd.res** and **xmservd.cf** can exist in two directories. The **xmservd** daemon first attempts to locate the files in the directory **/etc/perf**. Because the **/etc** directory is always unique for all hosts, even when some hosts are diskless hosts, defaults can be defined on a per-host basis. If one or both files are found in this directory, that file is used.

If a file is not found, it is searched for in **/usr/lpp/perfagent**. If the file cannot be located in this directory either, the **xmservd** continues without any of the actions that can be started from the file.

The **xmservd** usually produces a log, which alternates between two file names that are identical, apart from one ending in 1 and the other in 2. The file name of the log files are as follows:

- **/etc/perf/xmservd.log1**
- **/etc/perf/xmservd.log2**

Two more files are created by **xmservd** under certain circumstances:

- **/etc/perf/xmservd.mib**
- **/etc/perf/xmservd.state**

The **xmservd.mib** file is created whenever the **xmservd** daemon was started with the xmservd/SMUX interface active and subsequently sent a **SIGINT** signal (kill -2).

The **xmservd.state** file is created when the **xmservd** daemon is killed or aborts.

Neither of the four output files are created in other places than specified above.

# Explaining the xmperf Configuration File

This section explains the format of those lines in the **xmperf** configuration file that you use to define consoles and instruments. This is described in “Defining Consoles,” and “Defining Skeleton Consoles” on page 276 explains how to convert ordinary console definitions into skeleton console definitions.

The command line option **-v** is provided to assist you in debugging changes you make to the configuration file. It prints all configuration file lines to the **xmperf** log file. Lines with errors are followed by a line that begins with *** and explains the error or inconsistency.

## Defining Consoles

All console definition lines in the configuration file must have an identifier. This identifier is divided in four parts by periods (full stops). A few sample console definition lines are shown in the following example:

```
monitor.Mini   Monitor.1.width:          180
monitor.Mini   Monitor.1.height:         340
monitor.Mini   Monitor.1.x:              1108
monitor.Mini   Monitor.1.y:              580
monitor.Mini   Monitor.1.background:     black
monitor.Mini   Monitor.1.foreground:     grey70
```

The first element of the identifier is a keyword, which must be *monitor*. Next follows the name of the console you define. All lines with this same name are taken as part of the definition of the console, no matter where they appear in the configuration file. In the example above, the name of the defined console is “Mini Monitor.” You'll see that the name may have embedded blanks.

The third part of the identifier is used to identify the instrument within the console. The example above is thus defining part of the first instrument within the console “Mini Monitor.”

The fourth part of the identifier describes which property of the instrument is defined. It may be a single keyword, in which case it describes a property for the console itself or for an instrument as a whole. It may be followed by a period and a sequence number, in which case it describes a property for a value in an instrument. In all cases, the fourth element of the identifier must be followed by a colon.

After the colon comes the actual value of the property. In the preceding figure, for example, the first line sets the width of the console to 180 pixels while the last line sets the foreground color of instrument number 1 in the console to a color called “grey70.”

The keyword in the fourth part of the identifier determines whether the line defines a property for a console, an instrument, or a value. The following three sections describe the keywords that are valid for each of these groups.

The only keyword, which is required to define a console, is the input keyword defining a statistic to be plotted in an instrument of the console. All other keywords have defaults as detailed in the next sections.

## Console Keywords

Keywords that define console properties should always be defined using an instrument sequence number equal to the lowest instrument number in the console. It is strongly suggested that this always be sequence number 1. The valid keywords are:

**width** Defines the width in pixels of the window that will contain the console. Default is 400 pixels.

**height** Defines the height in pixels of the window that will contain the console. Default is 500 pixels.

**x** Defines the position of the left side of the console window, measured in pixels from the left side of the display. Default is position zero.

**y** Defines the position of the top side of the console window, measured in pixels from the top of the display. Default is position zero.

## Instrument Keywords

All keywords that describe instruments must be immediately followed by a colon and never by a sequence number. The valid keywords and related values are:

**left** Relative position of the left side of the instrument, given as a percentage of the width of the console window. Must be from 0 to 100 and at least 10 less than “right.” Default is 1.

**top** Relative position of the top side of the instrument, given as a percentage of the height of the console window. Must be from 0 to 100 and at least 10 less than “bottom.” Default is 1.

**right** Relative position of the right side of the instrument, given as a percentage of the width of the console window. Must be from 0 to 100 and at least 10 larger than “left.” Default is 99.

**bottom** Relative position of the bottom side of the instrument, given as a percentage of the height of the console window. Must be from 0 to 100 and at least 10 less than “top.” Default is 99.

**shift** Number of pixels to shift. Only used for recording graphs. Must be from one more than “space” to 20. Default is 4.

**space** Space between bars. Only used for bar graphs. Must be from 0 to one less than “shift.” Default is 2.

**history** Number of observations to keep in memory. Must be from 50 to 5,000. Default is 500.

**interval** Number of milliseconds between observations. Must be from 100 to 15,000. Default is 5000.

**background** Background color of instrument. Must be a color defined in the X color file. Default is black.

**foreground** Foreground color of instrument. Must be a color defined in the X color file. Default is white.

**backtile** Specifies the name of a tile (pixmap) that is used to paint the instrument. The tile name must be one of the following. The number in parentheses after the tile name indicates the number of the tile as shown in the color/tile dialog window of **xmperf**:

**foreground (1)**
Instrument is painted with 100% foreground and 0% background color.

**background (2)**
Instrument is painted with 0% foreground and 100% background color. This is the default tile.

**vertical (3)** Instrument is painted with a pattern that mixes foreground and background colors to produce a pattern of vertical lines.

**horizontal (4)** Instrument is painted with a pattern that mixes foreground and background colors to produce a pattern of horizontal lines.

**slant_right (5)**
Instrument is painted with a pattern that mixes foreground and background colors to produce a pattern of lines slanted to the right.

**slant_left (6)** Instrument is painted with a pattern that mixes foreground and background colors to produce a pattern of lines slanted to the left.

**plaid (7)** Instrument is painted with a pattern resembling the pattern in a plaid.

**triangles (8)** Instrument is painted with a pattern composed of triangles.

**wallpaper (9)** Instrument is painted with a pattern that resembles wallpaper.

**zigzags (10)** Instrument is painted with a zigzag pattern.

**fabric (11)** Instrument is painted with a pattern that looks somewhat like woven fabric.

**style** Defines the primary style of the instrument. Must be one of the following values:

**line** Line graph (default)

**area** Area graph

**skyline**
Skyline graph

**bar** Bar graph

**level** State bar graph

**light** State light graph

**pie** Pie chart

**meter** Speedometer graph

Actually, only the first three characters of the property name are used, because three characters are enough to make the graph type unique.

**stacked** Specifies whether stacking is to be used for values that use the primary style. Specify True if you want stacking to be used; otherwise specify False. Default is False.

## Value Keywords

This section describes how to define values for non-skeleton consoles. To see how to define values for skeleton consoles, see "Defining Skeleton Consoles" on page 276. The keywords used to define values must be followed by a period and the sequence number of the value. The value sequence number determines the sequence in which values are plotted, and influences the visual results when stacking is in use.

Plotting for each observation is done so that the lowest sequence number is plotted first. In the case of stacking, this means that the value with the lowest sequence number is plotted relative to zero. The next value is plotted relative to the previously plotted value, and so forth.

The keywords (here all shown with the sequence number 1, but it could be from 1 to 24) are:

**input.1** The path name of the value to be plotted. For non-skeleton consoles, this must always be a fully qualified path name (no wildcards). Below are a few examples of the use of this keyword:

```
monitor.Mini
Monitor.1.input.4:  IP/NetIF/tr0/ipacket
monitor.Mini Monitor.1.input.20:  Disk/hdisk0/busy
monitor.Maxi Monitor.7.input.18:  hosts/birte/Proc/runque
```

The third of the above lines specifies a path name qualified with a host name. Lines with such path names are bound to a specific host. The other two lines do not bind to any specific host. The **xmperf** program assumes that such lines refer to the host defined through the concept of *Localhost* as described in "The Meaning of Localhost in xmperf" on page 23.

For process statistics, where the statistic name includes the process ID, a tilde, and the name of the executing program, you can specify either the process ID followed by the tilde, or the name of the executing program. The example below shows how to specify a statistic for the *wait* pseudo process. The wait pseudo process always has a process ID of 514 on AIX 4.3.2. Both lines point to the same statistic.

```
monitor.Wait Monitor.1.input.1:  Proc/514~/usercpu
monitor.Wait Monitor.1.input.2:  Proc/wait/usercpu
```

If you specify a name of a program currently executing in more than one process, only the first one encountered will be found.

**color.1** The color used to plot the value. Must be a color defined in the X color file. Default is generated from a table of default values for the **ValueColor1** through **ValueColor24** X resources, depending on the sequence number of the value.

**tile.1** Specifies the name of a tile (pixmap) that is used to paint the value if the style of the value is neither line nor skyline and if the style of the instrument is not state light. The tile name must be one of the following. The number in parentheses after the tile name indicates the number of the tile as shown in the color/tile dialog window:

**foreground (1)**
Value is painted with 100% value color and 0% background color. This is the default tile.

**background (2)**
Value is painted with 0% value color and 100% background color.

**vertical (3)** Value is painted with a pattern that mixes value color and background colors to produce a pattern of vertical lines.

**horizontal (4)** Value is painted with a pattern that mixes value color and background colors to produce a pattern of horizontal lines.

**slant_right (5)**
Value is painted with a pattern that mixes value color and background colors to produce a pattern of lines slanted to the right.

**slant_left (6)**
Value is painted with a pattern that mixes value color and background colors to produce a pattern of lines slanted to the left.

**plaid (7)**
Value is painted with a pattern resembling the pattern in a plaid. This pattern is not suited for instruments with low *shift* values because it requires some space to be recognizable.

**triangles (8)**
Instrument is painted with a pattern composed of triangles.

**wallpaper (9)**
Instrument is painted with a pattern that resembles wallpaper.

**zigzags (10)**
Instrument is painted with a zigzag pattern.

**fabric (11)**
Instrument is painted with a pattern that looks somewhat like woven fabric.

**range.1**
The scale (range) used to plot the value. Given as two values separated by a dash. Default is supplied from the SPMI data repository. A couple of examples:

```
monitor.Mini Monitor.1.range.1:  0-100 monitor.Mini Monitor.1.range.18:
0-8
```

**thresh.1**
Threshold value. Used only by the state light graph. Default is zero.

**descending.1**
Type of threshold value. Used only by the state light graph. Specify True if the light must go on when the current value is below the threshold value. If you want the light to go on when the value is above the threshold, specify False or don't specify the keyword. Default is False.

**label.1**
Defines the user-specified text that is used to label the value in the instrument. Default is no user-specified text (the value path name is used).

**style.1**
Defines the secondary style for the value defined. Only valid for recording graphs. Must be one of the following values:

**line**
Line graph

**area**
Area graph

**skyline**
Skyline graph

**Bar**
Bar graph

The default is chosen as the same as the graph style of the instrument.

## Defining Skeleton Consoles

Skeleton consoles are defined like any other console with two exceptions. Neither the keywords defining the console, nor those defining the instruments, are different. The only difference is in the keyword used to define the values in the instruments of the console. The keyword that's different is the **input.1** keyword, which must be changed to **all.1** or **each.1**.

The difference is that the path name of the value must contain exactly one wildcard, and that the path of all the **all.1** and **each.1** keywords in the console must be the same up to, and including the wildcard.

Whether you use **all.1** or **each.1** for the keyword depends on what type of skeleton you want. See "Skeleton Instruments" on page 14 for an explanation of the two types of skeletons.

The following are three examples of sets of skeleton definitions:

```
monitor.Single-host Monitor.3.each.1: hosts/*/CPU/kern
monitor.Single-host Monitor.3.each.2: hosts/*/Syscall/total

monitor.Remote Mini Monitor.1.each.4: IP/NetIF/*/ipacket monitor.Remote Mini
```

```
Monitor.1.each.5: IP/NetIF/*/opacket

monitor.Uziza Disk Monitor.1.all.21: hosts/uziza/Disk/*/busy
```

The last line binds the skeleton to a specific host. When this is done, all value definitions in the console must be bound to the same host. If no host binding is done (as in the first two sets above), the concept of *Localhost* as described in "The Meaning of Localhost in xmperf" on page 23 applies.

**Note:** You can mix skeleton types within a console; just remember that all paths up to the wildcard must be the same, not only in an instrument but for all instruments in a console.

As mentioned previously, skeleton instruments of type "all" can only have one value defined. Thus, all values in the instantiated instrument will have the same color, namely as defined for the value in the skeleton instrument. Not only can it be dull, it effectively restricts the "all" type skeletons to use the state bar graph type. Otherwise you wouldn't be able to tell one value from another.

To cope with this, you can define the color for a value in a skeleton instrument of type "all" as **default**. This causes **xmperf** to allocate colors to the values dynamically as values are inserted during instantiation of the skeleton. Below is an example of a full value definition using this feature:

```
monitor.Processes.1.all.1: hosts/myhost/Proc/*/kerncpu
monitor.Processes.1.color.1: default
monitor.Processes.1.range.1: 0-100
monitor.Processes.1.label.1: cmd
```

## Defining Default Consoles

When **xmperf** is started, you can have one or more consoles opened automatically. This is done by adding one line to the configuration file for each console you want to be automatically opened. The following shows an example of such a line:

```
monitor.Mini Monitor.default
```

The line states the name of the default console, in this case "Mini Monitor," followed by the keyword *default*. More than one such line may exist in the configuration file. One console is opened for each such line. The consoles are opened sequentially in the same order as they appear in the configuration file.

## The xmperf Resource File

The X Window System resource file for **xmperf** defines resources you can use to enhance the appearance and behavior of **xmperf** and is installed as **/usr/lib/X11/app-defaults/XMperf**.

## Resources Defining Appearance

```
  #  xmperf  options
  #
  *GraphFont:
       -ibm-block-medium-r-normal--15-1
00-100-100-c-70-iso8859-1

  *background:                           grey70
  *XMMenubar.background:                 #cdb54d
  *XmCascadeButton.background:           #cdb54d
  Console*XMMenubar.background:          pink
  Console*XmCascadeButton.background:    pink
  Console*XMMenubar.foreground:          black
  Console*XmCascadeButton.foreground:    black
  *XMProcMenubar.background:             blue
  *XMProcList*XmCascadeButton.background:   blue
  *XMProcList*XmCascadeButton.foreground:   turquoise
  *XMSkelMenubar.background:             blue
  *XMSkelList*XmCascadeButton.background:   blue
```

```
*XMSkelList*XmCascadeButton.foreground:   turquoise
*XMHostMenubar.background:                blue
*XMHostList*XmCascadeButton.background:   blue
*XMHostList*XmCascadeButton.foreground:   turquoise
*XMHelpMenubar.background:                ForestGreen
*XMHelp*XmCascadeButton.background:       ForestGreen
*XMHelp*XmCascadeButton.foreground:       yellow
*XMColorWindow.background:                black
*XMTabWindow.background:                  grey70
*XMTabWindow.foreground:                  black
#
#    Dialog  Colors
#
*XMessage.background:    #eaeaad
*XMessage.foreground:    black
*XMOptions.background:   medium  aquamarine
*XMOptions.foreground:   black
*XMDelete.background:    blue
*XMDelete.foreground:    yellow
*XMExit.background:      firebrick
*XMExit.foreground:      black
*XMChanged.background:   yellow
*XMChanged.foreground:   black
*XMHelp.background:      DarkGreen
*XMHelp.foreground:      MediumSpringGreen
*XMStop.background:      pink
*XMStop.foreground:      black
*XMMsgbox.background:    DarkGreen
*XMMsgbox.foreground:    light grey
#
#    Light  Colors
#
*XMLight.background:     #729fff
*XMLight.foreground:     black
#
#    Digital  Clock  Colors
#
*XMSeekTime.foreground:  red
*XMSeekTime.background:  black
*XMPlayTime.foreground:  yellow
*XMPlayTime.background:  black
```

The sample file first defines the font used for all windows. The resource name to define the font specifically for Performance Toolbox for AIX is **GraphFont**. If you don't define this resource, **xmperf** tries to get a font name from the following resources:

- **graphfont**
- **FontList**
- **fontList**
- **Font**
- **font**

If none are defined, a suitable fixed-width font is used.

The next line defines the default color for all widgets with the name **XMMenubar**, which happens to be the menu bar in the main window and in the console windows when pull-down menus are used. The line defines the background color as a yellowish color.

The third line defines the background color for all widgets of the class **XmCascadeButton** to be the same yellowish color. This effectively paints all menu lines representing a cascade menu with this color.

The next four lines define the menu bar and cascading menu items as having foreground and background colors different from the defaults in consoles that are created when **xmperf** runs with pull-down menus. All consoles use the name **Console** for their top-level widget.

Ending this group of resource definitions are four sets of definitions that refer to special widget names. In all cases, resources are set that override menu bar and cascade menu colors for specific windows. The first six letters of each resource name tells which windows it's related to:

**XMProc**
: List of processes, whether used to instantiate skeleton consoles or selected from the Controls menu.

**XMSkel**
: Dialog boxes used to instantiate skeleton consoles except when wildcard is process or remote host.

**XMHost**
: Dialog box used to select from a list of hosts.

**XMHelp**
: Help windows.

Finally, the background color is defined for the window you use to select colors and tiles, and both foreground and background color is set for tabulating windows.

The next group of resources contains definitions of colors for seven distinct uses of dialog windows. The first one (XMessage) sets the colors for the main window used to display messages from **xmperf**. The next pair of lines (XMOptions) sets colors for all dialog windows used to change the configuration of instruments and consoles. The next three line pairs are used for dialog windows that pop up when you delete something, when you exit the program, and when you are warned that something has changed and you may want to save the changes. The last three pairs of resources define fore- and background colors for help windows, for the dialog box that warns about slow resynchronizing of remote instruments, and for general informational message boxes.

The two lines shown under the group "Light Colors" define the colors to use for the widgets containing state light instruments.

Finally, in the last group of resources, four lines define colors to use in the "digital clocks" used to seek in a recording file and show the playback time, respectively.

The colors used are examples and are chosen to make it easy for you to know what type of dialog window you see. The color difference makes it less likely that you confuse one type of dialog with another.

## Resources Defining Default Colors

Whenever a new value is added to an instrument, a default color is assigned to the value based upon the sequence number of the value within the instrument. The default colors are defined through resources. The following example shows the definition of default colors for values as supplied in the sample resource file.

```
#    Default  Value  Colors
#
*ValueColor1:   ForestGreen
*ValueColor2:   Goldenrod
*ValueColor3:   red
*ValueColor4:   MediumVioletRed
*ValueColor5:   LightSteelBlue
*ValueColor6:   SlateBlue
*ValueColor7:   green
*ValueColor8:   yellow
*ValueColor9:   BlueViolet
```

```
*ValueColor10:  SkyBlue
*ValueColor11:  pink
*ValueColor12:  GreenYellow
*ValueColor13:  SandyBrown
*ValueColor14:  orange
*ValueColor15:  plum
*ValueColor16:  MediumTurquoise
*ValueColor17:  LimeGreen
*ValueColor18:  khaki
*ValueColor19:  coral
*ValueColor20:  magenta
*ValueColor21:  cyan
*ValueColor22:  salmon
*ValueColor23:  sienna
*ValueColor24:  blue
```

## Execution Control Resources

The following example shows most of the resources that can be used to control the program execution. If a True or False option is set to True with an X resource, it cannot be overridden by a command line option.

```
#    Execution  Options
#
*LegendAdjust:      false
*LegendWidth:       14
*TabColumnWidth:    9
*TabWindowLines:    20
*DecimalPlaceLimit: 10
*GetExceptions:      false
*MonoLegends:       false
*PopupMenus:        false
*DirectDraw:        false
*Averaging:         100
*ScaleLines:        front
*BeVerbose:         false
```

All resources can be specified as command line options rather than resources, except for the following:

```
ScaleLines
DecimalPlaceLimit
MonoLegends
TabColumnWidth
TabWindowLines
```

The following is a list of all execution control resources defined for **xmperf**:

**ConfigFile**

Must be followed by a file name of a configuration file (environment) to be used in this execution of **xmperf**. If this resource is not given, the configuration file name is assumed to be **$HOME/xmperf.cf**. If this file does not exist, the file is searched for as described in Appendix B, “Performance Toolbox for AIX Files,” on page 271.

This resource can alternatively be specified by the command line argument **-o**.

**LegendAdjust**

If this resource is set to True, the size of value path names to display in instruments is adjusted to what is required for the longest path name in each instrument. The length may be less than the default fixed length (or the length specified by the **-w** option if used or the **LegendWidth** resource) but never longer than that. Note that the use of this option may result in instruments with time scales that are not aligned.

For pie chart graphs, adjustment is always done, regardless of the setting of this resource.

This resource can alternatively be specified by the command line argument **-a**.

**LegendWidth**
Must be followed by a number between 8 and 32 to define the number of characters from the value path name to display in instruments. The default number of characters is 12.

This resource can alternatively be specified by the command line argument **-w**.

**TabColumnWidth**
Must be followed by a number from 5 through 15 to define the width in characters of each column displayed in a tabulating window. The default width is 9.

**TabWindowLines**
Must be followed by a number from 2 through 100 to define the number of lines displayed in a tabulating window. The default line count is 20. If more than 25 lines are specified, tabulating windows have a vertical scrollbar to allow you to see all the detail lines in the window.

**DecimalPlaceLimit**
Defines the limit that determines if a data value is displayed with or without a decimal place in tabulating windows and in the state bar graph type. If the upper range defined for the value is less than or equal to this value, a decimal place is displayed. Otherwise no decimal place is displayed. The default upper range limit is 10.

**GetExceptions**
If this resource is set to True, **xmperf** requests all its data suppliers to forward exceptions. If this resource is set to False, data-supplier hosts will not forward exceptions to this invocation of **xmperf**.

This resource can alternatively be specified by the **-x** command line argument.

**MonoLegends**
If this resource is set to True, the text describing values in instruments with a primary style of line, skyline, area, bar, state bar, and pie is drawn in the instrument's foreground color. If this resource is set to False, the text is drawn in the value's color.

This resource cannot be specified by a command line argument.

**PopupMenus**
If this resource is set to True, popup menus are used rather than the pull-down menus. As described in Chapter 3, "The xmperf User Interface," on page 29, the overall menu structure may be based upon pull-down menus (which is the default) or popup menus as activated by setting this resource True. Pull-down menus may be easier to understand for occasional users, although popup menus generally provide a faster but less intuitive interface.

This resource can alternatively be specified by the command line argument **-u**.

**DirectDraw**
Usually, **xmperf** first draws graphical output to a pixmap and then, when all changes are done, moves the pixmap to the display. Generally, with a locally attached color display, performance is better when graphical output is redrawn from pixmaps, which is why this is the default. Also, a flaw in some levels of X Window System can be bypassed when this option is in effect. For monochrome displays and X stations, you may want to set this resource to True, which causes **xmperf** to draw graphical output directly to the display rather than always redrawing from a pixmap.

This resource can alternatively be specified by the command line argument **-z**.

**Averaging**
If this resource is set to a value greater than 25 and smaller than 100, averaging is activated. Averaging causes an "averaging" or "weighting" of all observations for state graphs before they are plotted. The number assigned to the resource is taken as the "weight percentage" to use when averaging the values plotted in state graphs. The formula used to calculate the average is:

```
val = new * weight/100 + old *
(100-weight) / 100
```

where:

*val* Is the value used to plot.

*new* Is the latest observation value.

*old* Is the *val* calculated for the previous observation.

*weight* Is the weight specified by the resource.

If a number outside the valid range is specified, averaging is not activated. This resource can alternatively be specified by the command line argument **-p**.

The weight is also used to calculate the weighted average line in tabulating windows.

**ScaleLines**

If this resource has a value of **back** or **front**, all recording graphs will have a horizontal, stippled line drawn at the 50% mark and the 100% mark. If the resource value is **back**, the lines are drawn before the values are plotted so that the actual graph is overlaying the scale lines. If the resource value is **front**, the lines are superimposed on top of the plotted values and will always be visible.

The default value for this resource is **back**. If the resource is set to any value but the two valid ones, scale lines are not drawn. The resource does not have corresponding command line arguments.

**BeVerbose**

If this resource is set True, configuration file lines are printed to the log file as they are processed. Any errors detected for a line are printed immediately below the line. This option is intended as a help to find and correct errors in the configuration file. Use this option if you don't understand why a line in your configuration file does not have the expected effect.

This resource can alternatively be specified by the command line argument **-v**.

## The azizo Resource File

The X Window System resource file for **azizo** defines resources you can use to enhance the appearance and behavior of **azizo** and is installed as:

`/usr/lib/X11/app-defaults/Azizo.`

```
#    General  azizo  options
#
*GraphFont:        -ibm-block-medium-r-normal--15-100-100-100-c-70-iso8859-1
*background:              grey70
*MetricHeight:            40
*MetricWidth:             600
*MetricLineDouble:        False
*MetricLinePlot:          False
*MainGraphCount:          16
*GraphWindowWidth:        862
*OnlyMetricColorIndex:    6
*FirstMetricColorIndex:   3
*SecondMetricColorIndex:  7
#
#        Major    screen    colors
#
*AZMain.background:                      grey70
*AZMetrics.background:                   grey70
MainGraph*XmDrawingArea.background:      black
MainGraph*XmLabel.background:            black
#
#        Dialog    Colors
#
*XMHelpMenubar.background:               ForestGreen
*XMHelp*XmCascadeButton.background:      ForestGreen
*XMHelp*XmCascadeButton.foreground:      yellow
*XMHelp.background:                      DarkGreen
```

```
*XMHelp.foreground:                     MediumSpringGreen
*InfoWindow.background:                 grey70
*AZAction.background:                   grey70
*AZMessage.background:                  grey70
*AZMessage.foreground:                  black
*ExitWindow.background:                 red
*ExitWindow.foreground:                 white
*PromptWindow.background:               LightSteelBlue
*PromptWindow.foreground:               black
*QuestionWindow.background:             goldenrod
*QuestionWindow.foreground:             black
*XMMsgbox.background:                   medium aquamarine
*XMMsgbox.foreground:                   black
```

The sample file first defines the default background color for all windows whenever this color is not overridden by some later resource setting. It then sets the font used for all windows. The resource name to define the font specifically is **GraphFont**. If you don't define this resource, **azizo** tries to get a font name from the following resources:

- **graphfont**
- **FontList**
- **fontList**
- **Font**
- **font**

If none are defined, a suitable, fixed-pitch font is used. Other X resources are, in alphabetical order:

**AZAction**
: The name of the widgets used to create action buttons. Use to set background color of the buttons.

**AZMain**
: The name of the base widgets used to create the main window. By referencing the **AZMain** widget name, you can set the foreground and background color of the main window.

**AZMessage**
: The name of the widget used to create the message window shown below the metrics selection window. Use to set the colors of the message window.

**AZMetrics**
: The name of all widgets used to create the metrics selection window. By referencing the **AZMetrics** widget name, you can set the foreground and background color of the metrics selection window and the metrics graphs. The color you assign for foreground becomes the default color for drawing the metrics graphs. It can be overridden by other X resources as explained below. To set the background color of the metrics selection window, use ***AZMetrics.background** as the resource name.

**BrightenFactor**
: Default brighten factor to use in the Print Box. Default is 100; permitted range is 0 - 200.

**ExitWindow**
: Can be used to give the exit dialog box a different color to make it stand out from other dialog boxes.

**FirstMetricColorIndex**
: This resource specifies the index into a table of defined **ValueColor1** through **ValueColor24** resources. The color selected by the index is used to draw the maximum value line of metrics in the metrics selection window when both maximum and minimum values are drawn. Default is foreground color of the metrics selection window as set by the **AZMetrics** resource.

**GraphWindowWidth**
The initial width of main graph windows. This width is the width of the window itself, not the main graph area alone. Default is 862 pixels.

**InfoWindow**
The name of all widgets used to create information windows. By referencing the **InfoWindow** widget name, you can set the foreground and background color of all information windows.

**HorizontalMargin**
The default horizontal margin for the Print Box. Default is 0.5 inch.

**MainGraph**
The name of all the widgets used to create main graphs is **MainGraph**. By assigning color values to the X resources defined as **MainGraph*XmLabel.background** and **MainGraph*XmLabel.foreground**, the appearance of the metrics label part can be changed. Similarly, the colors used in the graph part can be changed with the resources **MainGraph*XmDrawingArea.background** and **MainGraph*XmDrawingArea.foreground**. The supplied X resource file for **azizo** shows the setting of the background color for both parts of the main graph window.

**MainGraphCount**
Defines the maximum number of metrics that will be part of the top-level main graph after a new recording file has been read in. Defaults to 16 metrics.

**MainGraphHeight**
Sets the initial height (in pixels) of all main graphs. Default is 300 pixels; permitted range is 50 - 1,000 pixels.

**MainGraphWidth**
Sets the initial width (in pixels) of all main graphs except the top-level main graph. Default is 600 pixels; permitted range is 100 - 1,200 pixels. Note that main graphs are never scaled horizontally. If you increase the width of the main graph window beyond what is required to show the full graph width, empty space appears to the right of the graph. If you reduce the width of the main graph window to less than what is required to display the main graph, then a scrollbar is added to allow horizontal scrolling of the graph area.

**MainPlotStyle**
Sets the default plotting style for metrics in main graphs. Defaults to “average.” Permitted values are: “maximum,” “minimum,” “both,” and “average.”

**MetricHeight**
Sets the height in pixels of individual metrics graphs in the metrics selection window. Default is 40 pixels; permitted range is 10 - 200 pixels.

**MetricLegendWidth**
Sets the width of the area at the left side of a metrics graph that is used to display the metric path name. Default is 20 characters; permitted range is 8 - 32 characters.

**MetricLineDouble**
This resource can be set to either True or False. It is ignored if the resource **MetricLinePlot** is set to False. If both these resources are True, the default way of drawing individual metrics graphs is with two lines: one for the maximum and one for the minimum values. If this resource is set to False while **MetricLinePlot** is set to True, only the maximum values are drawn.

**MetricLinePlot**
This resource can be set to either True or False. If this resource and the resource **MetricLineDouble** are both set to True, the default way of drawing individual metrics graphs is with two lines: one for the maximum and one for the minimum values. If this resource is set to True while **MetricLineDouble** is set to False, only the maximum values are drawn. If this resource is set to False, individual metrics graphs are drawn in the “Bar, Max-to-Min” style as a series of vertical lines, each one connecting the maximum and minimum value in an interval.

**MetricWidth**
This resource sets the width of the individual metrics graphs in the metrics selection window. It also sets the width of the graph area of the top-level main graph. The default is 600 pixels. Specify in the range 100 through 1,000.

**MonoLegends**
If this resource is set True, all labels in main graphs are shown in the foreground color of the label part of the main graph. Probably not useful because it is not possible to see which line in the graph corresponds to the label.

**OnlyMetricColorIndex**
This resource is used to select the foreground color of a metrics graph when the style is not "Line, Max & Min." That is, when only one line is drawn in each metrics graph. The resource defines an index into a table of defined **ValueColor1** through **ValueColor24** resources to use for main graphs. Default is the foreground color of the metrics selection window as set by the **AZMetrics** resource.

**PaperHeight**
Default paper height to use in the Print Box. Default is 11 inches.

**PaperWidth**
Default paper width to use in the Print Box. Default is 8.5 inches.

**PrintCommand**
The default print command to use for all print operations. Default is **lp -d**.

**PromptWindow**
Can be used to give the prompt dialog boxes different foreground and background color to make them stand out from other dialog boxes. Prompt dialogs are those that request you to supply input before an action is performed.

**QuestionWindow**
Can be used to give the question dialog boxes different foreground and background color to make them stand out from other dialog boxes. Question dialogs are those that ask you to confirm or reject an action.

**SecondMetricColorIndex**
This resource specifies the index into a table of defined **ValueColor1** through **ValueColor24** resources. The color selected by the index is used to draw the minimum value line of metrics in the metrics selection window when both maximum and minimum values are drawn. Default is foreground color of the metrics selection window as set by the **AZMetrics** resource.

**VerticalMargin**
The default Vertical margin for the Print Box. Default is 0.5 inch.

**XMHelp**
Used to set the colors of help screens and their menus. See the figure "The azizo Resource File" on page 282 for examples.

**XMHelpMenubar**
Used to set the colors of help screen menu bars. See the figure "The azizo Resource File" on page 282 for an example.

**AZIcons**
The name of the widgets used to create the image of icons that indicate that a drop action is permitted. Use to set color.

**XMMsgbox**
The name of the widgets used to create message windows that inform the user of some condition. Use to set colors.

## Simple Help File Format

Several of the X Window System based programs in the Performance Toolbox for AIX share a common help file layout and have the same help function linked with the executables. These programs are **xmperf**, **exmon**, **azizo**, and **3dplay**.

The help function identifies a help screen from a string of characters, called the help ID. Only if a help text with this help ID is present in the help file for the program, can help be given. If no help text is found, a message box will inform the user of this.

When help is requested by a user, the help function is called with a help ID that depends on the way help is requested. The different ways to request help and how the help ID is retrieved for each of those is explained below:

*Help requested for xmperf console*
The help ID is the name of the console. If the console is an instantiated skeleton console, the help ID is assumed to be the name of the skeleton console, rather than the instance of it.

*Help requested for xmperf tools dialog*
The help ID is the string of characters that identify the tool as shown in the window frame of the tools dialog window.

*Help requested from other dialog box*
The help ID is hard-coded for the dialog box. In most cases, the help ID is identical to the text shown in the window frame of the dialog window.

*Help requested from menu*
The help ID is hard-coded for the menu item.

*Help requested from help index*
The help ID is the text shown in the selected help index line.

*Help requested for azizo action or object*
The help ID is hard coded for the action or object.

When a program that uses the simple help file starts, it looks in your home directory for a file with the same name as the program but with a file name extension of **.hlp**. If found, that file is used. If the file does not exist in your home directory, **xmperf** attempts to find it as described in Appendix B, “Performance Toolbox for AIX Files,” on page 271.

The format of a definition of a help screen is identical for all types of help texts. “Simple Help File Format” is an example of a help text for an **xmperf** console:

```
$help: Mini Monitor
This console contains a single state instrument. It is
the default console and is intended to be permanently
open as a monitor of important activity on the local system.
All values shown in the console are lively to experience
shorter or longer peaks during normal system operation.
```

The keyword **$help:** must appear exactly as shown (including the dollar sign and the colon) and must begin in column one. It must be followed by at least one byte of white space and the help ID.

Following the line that defines the help ID, you specify the help text exactly as you want it to appear in the help screen. When the text is displayed by the help function, the window is sized to match the line length and the number of lines in the help screen, up to 25 lines. If the help window has more than 25 lines, the scroll bar in the help window must be used to scroll the help text up and down.

The help ID (shown as “Mini Monitor” in the figure) identifies the help text and binds it to the situation where it is displayed. If the name matches a console name or tool name as defined in the configuration file, the help text is bound to the console or tool; otherwise the help text is assumed to be a general help text, possibly associated with a help button or a help menu item.

Each of the programs that use the help function has a set of hard coded help IDs. These are listed in the following sections:

## Predefined help IDs for xmperf

```
Change Value
Color Selection
Erase Recording File
Help on Help
History, number of observations
Host Selection
Interval, seconds
Main Window
Name of New Console
Primary Style and Stacking
Process Controls
Recording file exists
Remote Process List
Select Value
Shift, pixels per observation
Slow Resync
Space between bars
Wildcard Selection
```

## Predefined help IDs for exmon

```
Add Hosts
Command Execution
Delete Hosts
Delete Log
Exception Logs
Exit exmon
Files Changed
Help on Help
Read Log
```

## Predefined help IDs for azizo

```
Changing View Options
Config Icon
Configuration Exists
Configuration File
Configuration Line
Delete Configuration
Exit azizo
Exit Icon
Filter Icon
Help Icon
Help on Help
Info Icon
Information Window
Local Files Icon
Main Graph
Metrics Graph
Metrics Graph List
Metric Label in Main Graph
Pit Icon
Print Box
Print Icon
Replace Configuration
Report Box
Rescaling
Scale Icon
Select Recording File
View Icon
Writing Configuration
Writing Filtered Recording
Zoom-in
```

## Predefined help IDs for 3dplay

```
Erase Annotation File
Erase Recording File
Help on Help
Select 3dmon Recording File
```

# Appendix C. Performance Toolbox for AIX Commands

Following is a list of Performance Toolbox for AIX commands and operating system commands related to performance tuning:

**a2ptx** A program to generate recordings from ASCII files.

**azizo** The main program for analyzing recordings.

**chmon**
A sample program written to the data-consumer API.

**fdpr** A performance tuning utility for improving execution time and real memory utilization of user-level application programs.

**filemon**
Monitors the performance of the file system, and reports the I/O activity on behalf of logical files, virtual memory segments, logical volumes, and physical volumes.

**fileplace**
Displays the placement of file blocks within logical or physical volumes.

**filtd** The **filtd** programs allow you to define new statistics from existing ones through data reduction and alarms that are triggered by conditions you define and which can execute any command you desire.

**genkex**
The **genkex** command extracts the list of kernel extensions currently loaded onto the system and displays the address, size, and path name for each kernel extension in the list.

**genkld**
The **genkld** command extracts the list of shared objects currently loaded onto the system and displays the address, size, and path name for each object on the list.

**genld** The **genld** command extracts a list of loaded objects for each process currently running on the system.

**lockstat**
Displays simple and complex lock contention information.

**netpmon**
Monitors activity and reports statistics on network I/O and network-related CPU usage.

**ptxconv**
A program to convert between Performance Toolbox for AIX Version 1.1 to Version 2 or Version 1.2 recording file format.

**ptxmerge**
The **ptxmerge** program allows the user to specify up to 10 input files that are to be merged into one file.

**ptxrlog**
A program to create ASCII or binary recording files.

**ptxsplit**
A program to split recording files into multiple files.

**ptxtab** A program to tabulate the contents of recording files.

**rmss** Simulates a system with various sizes of memory for performance testing of applications.

**stripnm**
Displays the symbol information of a specified object file.

© Copyright IBM Corp. 1994, 2004

**svmon**
Captures and analyzes a snapshot of virtual memory.

**tprof** Reports CPU usage.

**xmpeek**
The **xmpeek** program allows you to ask any host about the status of its **xmservd** daemon.

**xmperf**
The **xmperf** program allows you to define monitoring environments to supervise the performance of the local system and remote systems.

**xmscheck**
When **xmservd** is started with the command line argument **-v**, its recording configuration file parser writes the result of the parsing to the log file.

**xmservd**
The **xmservd** daemon is always started from inetd. Therefore, command line options must be specified on the line defining **xmservd** to **inetd** in the file **/etc/inetd.conf**.

**3dmon**
The **3dmon** program is an X Window System-based program that displays statistics in a 3-dimensional graph where each of the two sides may have up to 24 statistics for a maximum of 576 statistics plotted— in a single graph.

**3dplay**
Beginning with Version 2.2., 3dplay is provided to play back 3dmon recordings in the same style that the data was originally displayed.

# 3dmon Command

The **3dmon** program is an X Window System based program that displays statistics in a 3-dimensional graph where each of the two sides may have up to 24 statistics for a maximum of 576 statistics plotted in a single graph.

To avoid clashes with X Window System command line options, never leave a blank between a command line option and its argument. For example, do not specify

```
3dmon -i 1 -p 75 -n
```

Instead, use:

```
3dmon -i1 -p75 -n
```

## Syntax

The **3dmon** program takes the following command line arguments, all of which are optional:

**3dmon** [**-v**ng] [-f *config_file*] [**-i** *seconds_interval*] [**-h** *hostname*] [**-w** *weight_percent*] [**-s** *spacing*] [**-p** *filter_percent*] [**-c** *config*] [**-a** "*wildcard_match_list*"] [**-t***resync_timeout*] [**-d** *invitation_delay*] [**-l** *left_side_tile*] [**-r** *right_side_tile*] [**-m** *top_tile*]

## Flags

**-v** Verbose. Causes the program to display warning messages about potential errors in the configuration file to stderr. Also causes **3dmon** to print a line for each statset created and for each statistic added to the statset, including the results of resynchronizing.

**-n** Only has an effect if a filter percentage is specified with the **-p** argument. When specified, draws only a simple outline of the grid rectangles for statistics with values that are filtered out. If not specified, a full rectangle is outlined and the numerical value is displayed in the rectangle.

**-g** Usually, **3dmon** will attempt to resynchronize for each statset it doesn't receive data-feeds for for

resync-timeout seconds. If more than half of the statsets for any host are found to not supply data-feeds, resynchronizing is attempted for all the statsets of that host. By specifying the **-g** option, you can force resynchronization of all the statsets of a host if any one of them becomes inactive.

**-f** Allows you to specify a configuration file name other than the default. If not specified, **3dmon** looks for the file **$HOME/3dmon.cf**. If that file does not exist the file is searched for as described in Appendix B, "Performance Toolbox for AIX Files," on page 271.

**-i** Sampling interval. If specified, this argument is taken as the number of seconds between sampling of the statistics. If omitted, the sampling interval is 5 seconds. You can specify from 1 to 60 seconds sampling interval.

**-h** Used to specify which host to monitor. This argument is ignored if the specified wildcard is "hosts." If omitted, the local host is assumed.

**Note:** With the Performance Toolbox Local feature of Version 2.2 or later, this flag always uses the local host name.

**-w** Modifies the default weight percentage used to calculate a weighted average of statistics values before plotting them. The default value for the weight is 50%, meaning that the value plotted for statistics is composed of 50 percent of the previously plotted value for the same statistic and 50 percent of the latest observation. The percentage specified is taken as the percentage of the previous value to use. For example, if you specify 40 with this argument the value plotted is:

```
.4 * previous + (1 - .4) * latest
```

Weight can be specified as any percentage from 0 to 100.

**-s** Spacing (in pixels) between the pillars representing statistics. The default space is 4 pixels. You can specify from 0 to 20 pixels.

**-p** Filtering percentage, **-p**. If specified, only statistics with current values of at least **-p** percent of the expected maximum value for the statistic are drawn. The idea is to allow you to specify monitoring "by exception" so statistics that are approaching a limit stand out while others are not drawn. Filtering can be specified as any percentage from 0 to 100. Default is 0%.

**-c** Configuration set. When specified, overrides the default configuration set and causes **3dmon** to configure its graph using the named configuration set. The argument specified after the **-c** must match one of the **wildcard** stanzas in the configuration file. If this argument is omitted, the configuration set used is the first one defined in the configuration file.

**-a** Wildcard match list. When specified, is assumed to be a list of host names. If the primary wildcard in the selected configuration set is **hosts**, then the list to display host names is suppressed as **3dmon** automatically selects the supplied hosts from the list of active remote hosts. Depending on the configuration set definition, **3dmon** then either goes directly on with displaying the monitoring screen or, when additional wildcards are present, displays the secondary selection list.

**Note:** With the Performance Toolbox Local feature of Version 2.2 or later, this flag always uses the local host name.

The list of host names must be enclosed in double quotation marks if it contains more than one host name. Individual host names must be separated by white space or commas.

The primary purpose of this option is to allow the invocation of **3dmon** from other programs. For example, you could customize NetView to invoke **3dmon** with a list of host names, corresponding to hosts selected in a NetView window.

**-t** Resynchronizing timeout. When specified, overrides the default time between checks for whether synchronizing is required. The default is 30 seconds; any specified timeout value must be at least 30 seconds.

**-d** Invitation delay. Allows you to control the time **3dmon** waits for remote hosts to respond to an

invitation. The value must be given in seconds and defaults to 10 seconds. Use this flag if the default value results in the list of hosts being incomplete when you want to monitor remote hosts.

**-l**
(Lowercase L). Specifies the number of the tile to use when painting the left side of the pillars. Specify a value in the range 0 to 8. The values correspond to the tile names:

- 0: foreground (100% foreground)
- 1: 75_foreground (75% foreground)
- 2: 50_foreground (50% foreground)
- 3: 25_foreground (25% foreground)
- 4: background (100% background)
- 5: vertical
- 6: horizontal
- 7: slant_right
- 8: slant_left

The default tile number for the left side is 1 (75_foreground).

**-r**
Specifies the number of the tile to use when painting the right side of the pillars. Specify a value in the range 0 to 8. The values correspond to the tile names specified above for option **-l**. The default tile number for the right side is 8 (slant_left).

**-m**
Specifies the number of the tile to use when painting the top of the pillars. Specify a value in the range 0 to 8. The values correspond to the tile names specified above for option **-l**. The default tile number for the top is 0 (foreground).

### Hardware Dependencies

On some graphics adapters in certain configurations, the **3dmon** program might not give you proper tiling. If you notice this, use the following command line arguments to suppress tiling:

```
3dmon -l0 -r0 -m0
```

Use the flags shown in addition to any other flags you may require. You can substitute the digit 4 for any of the zeroes shown above. The digit 0 means to paint the pillar in the foreground color; the digit 4 means to paint it in the background color.

## 3dplay Command

Beginning with Version 2.2., **3dplay** is provided to play back **3dmon** recordings in the same style that the data was originally displayed.

## Syntax

**3dplay** *RecordFile*>

## Parameters

| | |
|---|---|
| *Recordfile* | The name of a recording file created by **3dmon**. |

## Errors

If a non-**3dmon** recording file is provided as input, **3dplay** returns an error message and the recording file will not be played back.

## a2ptx Command

A program to generate recordings from ASCII files.

## Syntax

**a2ptx** *input_file output_file*

## Parameters

| | |
|---|---|
| *input_file* | This is a required parameter. |
| *output_file* | This is a required parameter. |

# azizo Command

The main program for analyzing recordings.

## Syntax

**azizo** [**-f** *recording_file*]

## Flags

The command line argument is optional and has the following meaning:

**-f** Recording file path name. Used to specify the name of a recording file to analyze. If the file is a valid recording file, **azizo** reads the file and processes it. If this argument is omitted or the specified file is not valid, **azizo** starts and awaits your selection of a recording file by clicking on the Local Files Icon.

Beginning with Version 2.2, this argument also brings up the recording's existing annotation file, or creates a new one, so that the user can take notes about the recording.

# chmon Command

Another sample program written to the data-consumer API is the program **chmon**. Source code to the program is in **/usr/samples/perfmgr/chmon.c**. The **chmon** program is also stored as an executable during the installation of the Manager component.

## Syntax

**chmon** [**-i** *seconds_interval*] [**-p** *no_of_processes*] [*hostname*]

## Parameters

| | |
|---|---|
| *seconds_interval* | Is the interval between observations. Must be specified in seconds. No blanks must be entered between the flag and the interval. Defaults to 5 seconds. |
| *no_of_processes* | Is the number of "hot" processes to be shown. A process is considered "hotter" the more CPU it uses. No blanks must be entered between the flag and the count field. Defaults to 0 (no) processes. |
| *hostname* | Is the host name of the host to be monitored. Default is the local host. |

The sample program exits after 2,000 observations have been taken, or when you type the letter `q` in its window.

## filtd command

The **filtd** programs allows you to define:

- New statistics from existing ones through data reduction.
- Alarms that are triggered by conditions you define and which can execute any command you desire.

### Syntax

The **filtd** program is designed to run as a daemon. It takes three command line arguments, all of which are optional:

**filtd** [**-f** *config_file*] [**-b** *buffer_size*] [**-p** *trace_level*]

### Flags

**-f** Overrides the default configuration file name. If this option is not given, the file name is assumed to be available in **/etc/perf/filter.cf** or else as described in Appendix B, "Performance Toolbox for AIX Files," on page 271. The configuration file is where you tell **filtd** what data reduction and alarm definitions you want.

**-p** Specifies the level of detail written to the log file. The trace level must be between 1 and 9. The higher the trace level the more is written to the log file. If this option is not specified, the trace level is set to zero.

**-b** Buffer size for communications with **xmservd** via RSI. The default buffer of 2048 bytes will allow for up to 60 statistics to be used in defining new statistics and alarms. If more are needed, the buffer size must be increased. It may also be necessary to increase the **xmservd** buffer size.

## ptxconv Command

A program to convert between **Performance Toolbox for AIX** Version 1.1 to Version 2 or Version 1.2 recording file format.

### Syntax

**ptxconv -v** {**1** | **2**} *input_file output_file*

### Flags

**v1** Converts a recording file with Version 2 or Version 1.2 format into the Version 1.1 format. This allows the use of an older version of **xmperf** to play the recording back.

**v2** Converts a recording file with Version 1.1 format into the later format. This allows any of the **Performance Toolbox for AIX** Version 1.2 and Version 2 programs that process recording files to work with the converted file.

*input_file* The path name of a recording file. The input file should be a file that was created with the version level the user wishes to convert from.

*output_file* The path name the user wishes the new recording file to have.

## ptxmerge Command

The **ptxmerge** program allows the user to specify up to 10 input files that are to be merged into one file. All files must be valid **Performance Toolbox for AIX** recording files in Version 2 format. When more than one input file is specified and one or more of the input files contain multiple sets of control information, only the records belonging to the first such set participate in the merge operation.

If only one input file is given, the program assumes you want it to rearrange the records in that file. If this file contains only one set of control information, then the output file is identical to the input file.

## Syntax

**ptxmerge** [ **-m**| **-p** *incr*|-t inc *r*] [ **-z**] *outfile input1* [ *input2* [ *input3*...]]

## Flags

**m**
: Only valid if exactly two input files are specified. Merges files, modifying all time stamps in the oldest file by the difference in time between the time stamps of the first value record in the two files.

**p**
: Only valid if exactly two input files are specified. Must be followed by the number of seconds to be added to all time stamps in the first input file before merging the files. This value may be negative.

**t**
: Only valid if exactly two input files are specified. Must be followed by the number of seconds to be added to all time stamps in the second input file before merging the files. This value may be negative.

**z**
: Optional. Preserves information about sets of statistics (statsets) when creating the resulting file. This is useful if the output file is to be used for playback with **xmperf**. The input files are merged together but each set of statistics are played back in instruments of the same contents (though not necessarily the same appearance) as the originals

# ptxrlog Command

A program to create ASCII or binary recording files.

## Syntax

**ptxrlog** {**-f** *infile*|**-m**|**-mf***infile*} [**-h** *hostname*] [**-i** *seconds*] [**-o** *outfile* [**-c**|**-s**|**-t**]|**-r** *binoutfile*] [**-l** *pagelen*] [**-b** *hhmm*] [**-e** *hh.mm*]

## Flags

**f**
: Name of a control file that contains a list of statistics to record. In the control file, each statistic must be given on a line on its own and with its full path name, excluding the host part, which is supplied by **ptxrlog** either from the **-h** argument or by using the local host name. If the **-f** argument is not given, the user is prompted for a list of statistics. If both the **-f** and the **-m** arguments are given, **ptxrlog** first selects the statistics given in the control file, then prompts the user to specify additional statistics.

**m**
: Manual input of statistic names. The user is prompted for a list of statistic names to be entered as full path names without the host part. The host part is supplied by **ptxrlog** either from the **-h** argument or by using the local host name. If both the **-f** and the **-m** arguments are given, **ptxrlog** first selects the statistics given in the control file, then prompts the user to specify additional statistics.

**h**
: Hostname of the host to monitor. This argument is used to identify the host to be monitored and, thus, to create the *hosts* part of the path names for the statistics to monitor. If this argument is not supplied, the host name of the local host is used.

**i**
: Sampling interval. Specifies the number of seconds between sampling of the specified statistics. If this argument is not supplied, the sampling interval defaults to 2 seconds.

**o**
: Output file name. Specify the name of the output file you want. If this argument is omitted, output goes to standard output and neither of the format flags **-c**, **-s**, or **-t** is permitted. If **-o** is given but neither of the three format flags is, the output looks the same as the output from **ptxtab** shown in the "Example of ptxtab Default Output Format" on page 100. The **-o** flag and the **-r** flag are mutually exclusive.

**c**
: The flag **-c** causes **ptxrlog** to format the output file as comma separated ASCII. The flag is only valid if **-o** is given. Each line in the output file contains one time stamp and one observation. Both fields are preceded by a label that describes the fields. The output looks the same as the **ptxtab** output shown in the "Example of ptxtab Comma Separated Output Format" on page 298. The flags **-c**, **-s**, and **-t** are mutually exclusive.

**s**
: The flag **-s** causes **ptxrlog** to format the output file in a format suitable for input to spreadsheet programs. The flag is only valid if **-o** is given. The output looks the same as the output from **ptxtab** output shown in the "Example of ptxtab Spreadsheet Output Format" on page 298. The flags **-c**, **-s**, and **-t** are mutually exclusive.

**t**
: Tab separated format. This flag is identical to the **-s** flag except that individual fields on the lines of the output file are separated by tabs rather than blanks. The flag is only valid if **-o** is given. The flags **-c**, **-s**, and **-t** are mutually exclusive.

**r** The **-r** flag specifies that the output from **ptxrlog** goes to a binary recording file in standard recording file format. The name of the output file must be specified after the flag. The **-o** flag and the **-r** flag are mutually exclusive.

**l** (Lowercase L) Specifies the number of lines per page when neither the **-o** nor the **-r** flag is specified or when the **-o** flag is specified but neither of the **-c, -s,** or **-t** flags is specified. If this flag is omitted, the output is formatted with 23 lines per page if the **-o** flag is omitted; otherwise with 65 lines per page. When the **-o** flag is given, a page eject is inserted at the beginning of each page.

**b** Begin recording. If this argument is omitted, **ptxrlog** begins recording immediately. The flag and arguments are used to start the recording at a specified later time. The flag must be followed by the start time in the format *hhmm*, where:

*hh* = Hour in 24 hour time (midnight is 00).

*mm* = Minutes.

**e** End recording. Specifies the number of hours and minutes recording must be active. The flag must be followed by the number of hours and minutes in the format *hh.mm*, where:

*hh* = Number of hours to record.

*mm* = Number of minutes to record.
If this argument is omitted, the recording continues for 12 hours. A maximum of 24 hours can be specified. When the time specified by this argument has elapsed, **ptxrlog** terminates.

### Binary Recording Files

When the **-r** flag is used, output is written to the file name specified after the flag. If the file exists when recording starts, it is opened for append. After opening the binary output file, whether for creation or append, **ptxrlog** writes the control records to the file. For existing files, this causes the file to contain more than one set of control records and may require you to process the file with **ptxmerge** or **ptxsplit** before you can process the file with **xmperf** or **azizo.**

### Resynchronizing by ptxrlog

The ptxrlog program initiates a resynchronizing with the data-supplier host if the data-supplier host sends an **i_am_back** packet. This usually happens if the data-supplier host's **xmservd** daemon has died and is restarted.

The **ptxrlog** initiates a resynchronizing with the data-supplier host if no **data_feed** packets have been received for ten times the specified sampling interval.

## ptxsplit Command

A program to split recording files into multiple files.

## Syntax

**ptxsplit** { **-p** *parts*| **-s** *size*| **-h**| **-b**| **-f** *cfile*| **-d** *hhmm* [ **-t** *dhhmm*]} *infile*

## Flags

The command line arguments are all mutually exclusive, except that the **-t** argument is only valid if the **-d** argument is given. One of the arguments must be specified. The arguments are:

**p** Split in parts of equal size. Must be followed by the number of parts the input file shall be divided into. The output files are approximately the same size and begin with a set of control records. The output file names are **infile.p1, infile.p2, ... infile.p***n*. Statsets are preserved in the output as are any console records.

**s** Split in parts of equal size. Must be followed by the size you want each output file to have. The output files, except the last one, usually are slightly smaller than the specified size; the last file may be much smaller. The output files all begin with a set of control records. The output file names are **infile.s1, infile.s2, ... infile.s***n*. Statsets are preserved in the output as are any console records.

**h** Split into files according to the host name of individual observations. The output files all begin with a set of control records. The output file names are **infile.hostname1, infile.hostname2, ... infile.hostname***n*. Statsets are preserved in the output. Any console records are discarded.

**b** Split into files for each set of control records encountered. The output files all begin with a set of control records. The output file names are **infile.b1, infile.b2, ... infile.b***n*. Statsets are preserved in the output as are any console records.

**f** Split into two files. The flag must be followed by a file name of a control file. The first output file is to contain all occurrences of the statistics listed in the control file. Remaining statistics are written to the second output file. Statistics are specified in the control file with their full path name. The control file may contain comment lines beginning with the character # (number sign). If the *hosts* part of the path name is omitted, statistics are selected across all host names. If the *hosts* part of the path name is supplied, an exact match is required for a statistic to be selected. The first output file has the name **infile.sel**, the second outfile is called **infile.rem**. Statsets are not preserved in the output files.

The program **ptxls** can produce a list of the statistics contained in a recording file. The output from the program has the format required for the control file. Use it by redirecting **ptxls** output to a file; then edit the file to include only the statistics you want in the file **infile.sel**.

**d** Split after duration into parts covering time periods of equal size. Must be followed by the duration span of each file, given as **hhmm**, where:

*hh* = Hours.

*mm* = Minutes.

If the **-t** argument is omitted, the time period begins with the earliest value record in the input file; otherwise with the time specified on the **-t** argument. The output files all begin with a set of control records. The output file names are **infile.d1, infile.d2, ... infile.d***n*. Statsets are preserved in the output as are any console records.

**t** Only valid if the **-d** argument is given. Specifies a point in time that shall be used to split the input file. Must be followed by a time in the format **dhhmm**, where:

*d* = Day of week, Sunday = day 0.

*hh* = Hours.

*mm* = Minutes.

The time given may lie outside the time period covered by the input recording file. If the time given differs from the time stamp of the first value record in the input file, the first output file contains data for an interval smaller than that requested with the **-d** argument.

For example, assume a recording file's first value record has a time stamp corresponding to 30830 (day 3, at 8:30 a.m.) and you invoke **ptxsplit** with the command line:

```
ptxsplit -d0600 -t00000 recording_file
```

This causes the first file to cover the interval from 8:30 a.m. until 11:59 a.m., the next one from 12:00 noon until 5:59 p.m., and so on until there are no more value records in the input file.

Consider splitting the same file with the command line:

```
ptxsplit -d0600 -t40800 recording_file
```

The **-t** argument, in this case, gives a point in time later than the first value record's time stamp. The program determines the time to place the first split point by stepping backwards in time from day 4 at 8:00 a.m. in steps of six hours (as per the **-d** argument) until it has passed the time stamp of the first value record. This would be on day 3 at 8:00 a.m. This is the reference point. The first output file covers day 3 from 8:30 a.m. to 1:59 p.m., the next from 2 p.m. to 7:59 p.m., and so forth.

## ptxtab Command

A program to tabulate the contents of recording files.

## Syntax

**ptxtab** [**-l** *lines* | **-c** | **-s**] [**-r** | **-t**] *recording_file*

## Flags

**l**
: The flag **-l** (lowercase L) is used to specify the number of lines per page you want the output files formatted for. The default is 23 lines per page, which is ideal for viewing the output in a 25-line window or on a terminal with 25 lines. If you specify 0 (zero) lines per page, pagination is suppressed. If the value is given as non-zero, it must be between 10 and 10,000. The flags **-l**, **-c**, and **-s** are mutually exclusive.

**c**
: The flag **-c** causes **ptxtab** to format the output files as comma separated ASCII. Each line in the output files contains one time stamp and one observation. Both fields are preceded by a label that describes the fields. An example of output formatted this way is shown in the "Example of ptxtab Comma Separated Output Format." The eight detail lines shown correspond to the first two detail lines in the "Example of ptxtab Default Output Format" on page 100. The flags **-l**, **-c**, and **-s** are mutually exclusive.

**s**
: The flag **-s** causes **ptxtab** to format the output files in a format suitable for input to spreadsheet programs. When this flag is specified, it is always assumed that the **-r** flag is also given. An example of formatting with the **-s** flag is shown in the "Example of ptxtab Spreadsheet Output Format." The detail lines shown correspond to the detail lines in the "Example of ptxtab Default Output Format" on page 100. This output format also matches the requirements of the **a2ptx** input file format. The flags **-l**, **-c**, and **-s** are mutually exclusive.

**r**
: The flag **-r** is independent of the other flags. It specifies that when *SiCounter* data is sent to the **ptxtab** output files, they are presented as rates per second. Without this option, **ptxtab** presents this data as the delta value in the interval. The flags **-r** and **-t**are mutually exclusive.

**t**
: The flag **-t** is independent of the other flags. It specifies that when SiCounter data is sent to the **ptxtab** output files, they are presented as absolute values. In other words, this flag causes SiCounter values to be treated as SiQuantity values. Without this option, **ptxtab** presents this data as the delta value in the interval. The flags **-r** and **-t** are mutually exclusive.

## Example of ptxtab Comma Separated Output Format

```
#Monitor: Nice Monitor --- hostname: nchris
Time="1994/01/07   15:36:03",   PagSp/%totalused=27.82
Time="1994/01/07   15:36:03",   PagSp/%totalfree=72.18
Time="1994/01/07   15:36:03",   Mem/Virt/pagein=8
Time="1994/01/07   15:36:03",   Mem/Virt/pageout=20
Time="1994/01/07   15:36:07",   PagSp/%totalused=27.82
Time="1994/01/07   15:36:07",   PagSp/%totalfree=72.18
Time="1994/01/07   15:36:07",   Mem/Virt/pagein=7
Time="1994/01/07   15:36:07",   Mem/Virt/pageout=17
```

## Example of ptxtab Spreadsheet Output Format

```
#Monitor: Nice Monitor --- hostname: nchris
"Timestamp"             "PagSp/%totalused" "PagSp/%totalfree" "Mem/Virt/pagein "Mem/Virt/pageout"
"1994/01/07  15:36:03"   27.8               72.2               8                20
"1994/01/07  15:36:07"   27.8               72.2               7                17
"1994/01/07  15:36:11"   27.8               72.2               3                283
"1994/01/07  15:36:15"   27.8               72.2               28               48
"1994/01/07  15:36:19"   28.2               71.8               56               41
"1994/01/07  15:36:23"   29.5               70.5               29               38
"1994/01/07  15:36:27"   31.5               68.5               0                62
"1994/01/07  15:36:31"   32.4               67.6               70               1
"1994/01/07  15:36:35"   32.6               67.4               73               32
"1994/01/07  15:36:39"   32.8               67.2               156              0
"1994/01/07  15:36:43"   34.5               65.5               167              4
"1994/01/07  15:36:47"   34.4               65.6               163              0
"1994/01/07  15:36:51"   31.1               68.9               12               57
"1994/01/07  15:36:55"   30.2               69.8               35               34
"1994/01/07  15:36:59"   28.0               72.0               15               0
"1994/01/07  15:37:04"   28.0               72.0               15               0
```

## xmpeek Command

The **xmpeek** program allows you to ask any host about the status of its **xmservd** daemon.

### Syntax

**xmpeek** [**-a**|**-l**] [*hostname*]

### Flags

| | |
|---|---|
| **-a** | If this flag is specified, one line is listed for each data consumer known by the daemon. If omitted, only data consumers that currently have instruments (statsets) defined with the daemon are listed. This flag is optional. |
| **-l** | (lowercase L) is explained in "Using the xmpeek Program to Print Available Statistics" on page 163. This flag is optional. |
| *host name* | If host name is specified, the daemon on the named host is asked. If no host name is specified, the daemon on the local host is asked. |

### Examples

The following is an example of the output from the **xmpeek** program:

```
Statistics for xmservd daemon on *** birte ***
Instruments currently defined:    1
Instruments currently active:     1
Remote monitors currently known:  2
--Instruments--- Values  Packets
                                Internet Protocol
Defined Active  Active  Sent        Address      Port Hostname
------- ------- ------- ------- ---------------- ---- ----------
1       1       16      3,344   129.49.115.208   3885 xtra
```

Output from **xmpeek** can take two forms.

The first form is a line that informs you that the **xmservd** daemon is not feeding any data-consumer programs. This form is used if no statsets are defined with the daemon and no command flags are supplied.

The second form includes at least as much as is shown in the preceding example, except that the single detail line for the data consumer on host **xtra** is shown only if either the **-a** flag is used or if the data consumer has at least one instrument (statset) defined with the daemon. Note that **xmpeek** itself appears as a data consumer because it uses the RSi API to contact the daemon. Therefore, the output always shows at least one known monitor.

In the fixed output, first the name of the host where the daemon is running is shown. Then follows three lines giving the totals for current status of the daemon. In the above example, you can see that only one instrument is defined and that it's active. You can also see that two data consumers are known by the daemon, but that only one of them has an instrument defined with the daemon in **birte**. Obviously, this output was produced without the **-a** flag.

An example of more activity is shown in the following sample output from **xmpeek**. The output is produced with the command:

```
xmpeek -a birte
```

Notice that some detail lines show zero instruments defined. Such lines indicate that an **are_you_there** message was received from the data consumer but that no states were ever defined or that any previously defined states were erased.

```
Statistics for smeared daemon on *** birte ***
   Instruments currently defined:    16
   Instruments currently active:     14
   Remote monitors currently known:   6
--Instruments--- Values Packets   Internet Protocol
 Defined Active  Active  Sent         Address      Port  Hostname
   8       8       35    10,232    129.49.115.203  4184   birte
   6       4       28     8,322    129.49.246.14   3211   umbra
   0       0        0         0    129.49.115.208  3861   xtra
   1       1       16     3,332    129.49.246.14   3219   umbra
   0       0        0         0    129.49.115.203  4209   birte
   1       1       16       422    129.49.115.208  3874   xtra
------- ------- ------- ----------
  16      14       95    22,308
```

Notice that the same host name may appear more than once. This is because every running copy of **xmperf** and every other active data-consumer program is counted and treated as a separate data consumer, each identified by the port number used for UDP packets as shown in the **xmpeek** output.

The second detail shows that one particular monitor on host **umbra** has six instruments defined but only four active. This would happen if a remote **xmperf** console has been opened but is now closed. When you close an **xmperf** console, it stays in the Monitor menu of the **xmperf** main window and the definition of the instruments of that console remains in the tables of the data-supplier daemon but the instruments are not active.

## xmperf Command

The **xmperf** program allows you to define monitoring environments to supervise the performance of the local system and remote systems.

### Syntax

**xmperf** [**-v auxz**] [**-w** *width*] [**-o** *options_file*] [**-p** *weight*] [**-h** *localhostname*] [**-r** *network_timeout*]

### Flags

All command line options are optional and all except **-r** and **-h** correspond to X Window System resources that can be used in place of the command line arguments. The options **v**, **a**, **u**, **x**, and **z** are true or false options. If one of those options is set through an X Window System resource, it cannot be overridden by the corresponding command line argument. The options are described as follows:

**v**
: Verbose. This option prints the configuration file lines to the **xmperf** log file **$HOME/xmperf.log** as they are processed. Any errors detected for a line will be printed immediately below the line. The option is intended as a help to find and correct errors in a configuration file. Use the option if you don't understand why a line in your configuration file does not have the expected effect.

  Setting the X Window System resource **BeVerbose** to true has the same effect as this flag.

**a**
: Adjust size of the value path name that is displayed in instruments to what is required for the longest path name in each instrument. The length can be less than the default fixed length (or the length specified by the **-w** option if used) but never longer. The use of this option can result in consoles where the time scales are not aligned from one instrument to the next.

  **Note:** For pie chart graphs, adjustment is always done, regardless of this command line argument.

  Setting the X Window System resource **LegendAdjust** to true has the same effect as this flag.

**u**
: Use popup menus. As described in "Console Windows" on page 33, the overall menu structure can be based upon pull-down menus (which is the default) or popup menus as activated with this flag. Typically, pull-down menus are easier to understand for occasional users; while popup menus provide a faster, but less intuitive interface.

  Setting the X Window System resource **PopupMenus** to true has the same effect as this flag.

**x** Subscribe to exception packets from remote hosts. This option makes **xmperf** inform all the remote hosts it identifies that they should forward exception packets produced by the **filtd** daemon, if the daemon is running. If this flag is omitted, **xmperf** will not subscribe to exception packets.

Setting the X Window System resource **GetExceptions** to true has the same effect as this flag.

**z** For monochrome displays and X stations, you might want to try the **-z** option, which causes **xmperf** to draw graphical output directly to the display rather than always redrawing from a pixmap. By default, **xmperf** first draws graphical output to a pixmap and then, when all changes are done, moves the pixmap to the display. Generally, with a locally-attached color display, performance is better when graphical output is redrawn from pixmaps. Also, a flaw in some levels of X Window System can be bypassed when this option is in effect.

Setting the X Window System resource **DirectDraw** to true has the same effect as this flag.

**w** Must be followed by a number between 8 and 32 to define the number of characters from the value path name to display in instruments. The default number of characters is 12.

Alternatively, the legend width can be set through the X Window System resource **LegendWidth**.

**o** Must be followed by a file name of a configuration file (environment) to be used in this execution of **xmperf**. If this option is omitted, the configuration file name is assumed to be **$HOME/xmperf.cf**. If this file is not found, the file is searched for as described in **Performance Toolbox for AIX** Files (Appendix B, “Performance Toolbox for AIX Files,” on page 271).

Alternatively, the configuration file name can be set through the X Window System resource **ConfigFile**.

**p** If given, this flag must be followed by a number in the range 25-100. When specified, this flag turns on “averaging” or “weighting” of all observations for state graphs before they are plotted. The number is taken as the “weight percentage” to use when averaging the values plotted in state graphs. The formula used to calculate the average is:

val = new * weight/100 + old * (100-weight) / 100

where:

*val* Is the value used to plot.

*new* Is the latest observation value.

*old* Is the *val* calculated for the previous observation.

*weight* Is the weight specified by the **-p** flag. If a number outside the valid range is specified, a value of 50 is used. If this flag is omitted, averaging is not used.

Alternatively, the averaging weight can be set through the X Window System resource **Averaging**.

The weight also controls the calculation of weighted average in tabulating windows.

**h** Must be followed by the host name of a remote host that is to be regarded as *Localhost*. The *Localhost* is used to qualify all value path names that do not have a host name specified. If not specified, *Localhost* defaults to the host where **xmperf** executes.

**Note:** With the Performance Toolbox Local feature of Version 2.2 or later, this flag always uses the local host name.

**r** Specifies the timeout (in milliseconds) used when waiting for responses from remote hosts. The value specified must be between 5 and 10,000. If not specified, this value defaults to 100 milliseconds.

**Note:** On networks that extend over several routers, gateways, or bridges, the default value is likely to be too low.

One indication of a too low timeout value is when the list of hosts displayed by **xmperf** contains many host names that are followed by two asterisks. The two asterisks indicate that the host did not respond to **xmperf** broadcasts within the expected timeout period. The “ Host Selection List from xmperf” shows how some hosts in a host selection list have asterisks. The list shown was generated in a network with multiple levels of routers where the default timeout is on the low side during busy hours.

## xmscheck Command

When **xmservd** is started with the command line argument **-v**, its recording configuration file parser writes the result of the parsing to the log file. The output includes a copy of all lines in the recording configuration file, any error messages, and a map of the time scale with indication of when recording starts and stops.

Although this is useful to document what is read from the recording configuration file, it is not a useful tool for debugging of a new or modified recording configuration file. Therefore, the program **xmscheck** is available to preparse a recording configuration file before you move it to the directory **/etc/perf**, where **xmservd** looks for the recording configuration file.

When **xmscheck** is started without any command line argument, it parses the file **/etc/perf/xmservd.cf**. This way, you can determine how the running daemon is configured for recording. If a file name is specified on the command line, that file is parsed.

Output from **xmscheck** goes to stdout. The parsing is done by the exact same module that does the parsing in **xmservd**. That module is linked in as part of both programs. The parsing checks that all statistics specified are valid and prints the time scale for starting and stopping recording in the form of a "time table."

In the time table, each minute has a numeric code. The meaning of codes is as follows:

0 Recording is inactive. Neither a start nor a stop request was given for the minute.
1 Recording is active. Neither a start nor a stop request was given for the minute.
2 Recording is inactive. A stop request was given for the minute.
3 Recording is active. A start request was given for the minute.

The following example shows how **xmscheck** formats the time table. Only the part of the table that covers Tuesday is shown. The example shown in Example of Recording Configuration File (in the "Hot Lines (xmservd Recording Only)" on page 172 section) was used to produce this output.

```
Day 2, Hour 00: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 01: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 02: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 03: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 04: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 05: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 06: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 07: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 08: 000000000000000000000000000000311111111111111111111111111111
Day 2, Hour 09: 111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 10: 111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 11: 111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 12: 200000000000000000000000000000000000000000000000000000000000
Day 2, Hour 13: 311111111111111111111111111111111111111111111111111111111111
Day 2, Hour 14: 111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 15: 111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 16: 111111111111111111111111111111111111111111111111111111111111
Day 2, Hour 17: 200000000000000000000000000000000000000000000000000000000000
Day 2, Hour 18: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 19: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 20: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 21: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 22: 000000000000000000000000000000000000000000000000000000000000
Day 2, Hour 23: 000000000000000000000000000000000000000000000000000000000000
```

## xmservd Command

The **xmservd** daemon is always started from **inetd**. Therefore, command line options must be specified on the line defining **xmservd** to **inetd** in the file `/etc/inetd.conf`.

## Syntax

**xmservd** [**-v**] [**-b** *UDP_buffer_size*] [**-i** *min_remote_interval*] [**-l** *remove_consumer_timeout*] [**-m** *supplier_timeout*] [**-p** *trace_level*] [**-s** *max_logfile_size*] [**-t** *keep_alive_limit*] [**-x** *xmservd_execution_priority*]

## Flags

All command line options are optional. The options are:

**v**
: Verbose. Causes parsing information for the **xmservd** recording configuration file to be written to the **xmservd** log file.

**b**
: Defines the size of the buffer used by the daemon to send and receive UDP packets. The buffer size must be specified in bytes and can be from 4,096 to 16,384 bytes. The buffer size determines the maximum number of data values that can be sent in one **data_feed** packet. The default buffer size is 4096 bytes, which allows for up to 124 data values in one packet.

**i**
: Defines the minimum interval in milliseconds with which data feeds can be sent. Default is 500 milliseconds. A value between 100 and 5,000 milliseconds can be specified. Any value specified is rounded to a multiple of 100 milliseconds. Whichever minimum remote interval is specified causes all requests for data feeds to be rounded to a multiple of this value. See further details in section “Rounding of Sampling Interval” on page 156.

**l**
: (Lowercase L). Sets the **time_to_live** after feeding of statistics data has ceased as described in section “Life and Death of xmservd” on page 157. Must be followed by a number of minutes. A value of 0 (zero) minutes causes the daemon to stay alive forever. The default **time_to_live** is 15 minutes.

  This value is also used to control when to remove inactive data-consumers as described in Removing Inactive Data-Consumers (“Removing Inactive Data Consumers” on page 158).

**m**
: When a dynamic data-supplier is active, this value sets the number of seconds of inactivity from the DDS before the SPMI assumes the DDS is dead. When the timeout value is exceeded, the **SiShGoAway** flag is set in the shared memory area and the SPMI disconnects from the area. If this flag is not given, the timeout period is set to 90 seconds.

  The size of the timeout period is kept in the SPMI common shared memory area. The value stored is the maximum value requested by any data consumer program, including **xmservd**.

**p**
: Sets the trace level, which determines the types of events written to the log file **/etc/perf/xmservd.log1** or **/etc/perf/xmservd.log2**. Must be followed by a digit from 0 to 9, with 9 being the most detailed trace level. Default trace level is 0 (zero), which disables tracing and logging of events but logs error messages.

**s**
: Specifies the approximate maximum size of the log files. At least every **time_to_live** minutes, it is checked if the currently active log file is bigger than **max_logfile_size**. If so, the current log file is closed and logging continues to the alternate log file, which is first reset to zero length. The two log files are **/etc/perf/xmservd.log1** and **/etc/perf/xmservd.log2**. Default maximum file size is 100,000 bytes. You cannot make **max_logfile_size** smaller than 5,000 or larger than 10,000,000 bytes.

**t**
: Sets the **keep_alive_limit** described in section “Life and Death of xmservd” on page 157. Must be followed by a number of seconds from 60 to 900 (1 to 15 minutes). Default is 300 seconds (5 minutes).

**x**
: Sets the execution priority of **xmservd**. Use this option if the default execution priority of **xmservd** is unsuitable in your environment. Generally, the daemon should be given as high execution priority as possible (a smaller number gives a higher execution priority).

  On systems other than IBM **RS/6000** systems, the **-x** flag is used to set the nice priority of **xmservd**. The nice priority is a value from -20 to 19. Default is -20.

# Appendix D. ARM Subroutines and Replacement Library Implementation

This appendix provides information about the following:

- "ARM Subroutines"
- "ARM Replacement Library Implementation" on page 314

For further information, see Chapter 17, "Response Time Measurement," on page 191.

## ARM Subroutines

This section describes the use and implementation of the PTX version of the ARM library.

### arm_init Subroutine

#### Purpose

The **arm_init** subroutine is used to define an application or a unique instance of an application to the ARM library. In the PTX implementation of ARM, instances of applications can't be defined. See "Implementation Specifics" on page 306. An application must be defined before any other ARM subroutine is issued.

#### Library

ARM Library (**libarm.a**).

#### Syntax

```
#include arm.h

arm_appl_id_t arm_init( arm_ptr_t     *appname,      /* application name
*/
       arm_ptr_t     *appl_user_id, /* Name of the application user  */
       arm_flag_t    flags,         /* Reserved = 0                  */
       arm_data_t    *data,         /* Reserved = NULL               */
       arm_data_sz_t data_size);    /* Reserved = 0                  */
```

#### Description

Each application needs to be defined by a unique name. An application can be defined as loosely or as rigidly as required. It may be defined as a single execution of one program, multiple (possibly simultaneous) executions of one program, or multiple executions of multiple programs that together constitute an application. Any one user of ARM may define the application so it best fits the measurement granularity desired. Measurements are always collected for each unique combination of:

1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

#### Parameters

**appname**

A unique application name. The maximum length is 128 characters including the terminating zero. The argument is converted to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This key is used to look for an application structure in the library's private shared memory area. If a structure is found, its use-count is incremented and the application ID stored in the structure is returned to the caller. If the structure is not found, one is created, assigned the next free application ID and given a use-count of one. The new assigned application ID is returned to the caller.

© Copyright IBM Corp. 1994, 2004 

Up-to 64 bytes, including the terminating zero, of the *appname* parameter is saved as the description of the **SpmiCx context** (“SpmiCx Structure” on page 206) that represents the application in the Spmi hierarchy. The key is used as the short name of the context.

**appl_user_id**

Can be passed in as NULL or some means of specifying a user ID for the application. This allows the calling program to define unique instances of an application. In the PTX implementation of the ARM API, this parameter is ignored. Consequently, it is not possible to define unique instances of an application. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes in length, including the terminating zero.

For the implementation to take this argument in use, another context level would have to be defined between the application context and the transaction context. This was deemed excessive.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

### Return Values

If successful, the subroutine returns an **appl_id** application identifier. If the subroutine fails, a value less than zero is returned.

### Error Codes

No error codes are defined by the PTX implementation of the ARM API.

### Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the implementation of **arm_init** doesn’t allow unique instances of an application to be defined. The **appl_id** associated with an application is stored in the ARM shared memory area and will remain constant throughout the life of the shared memory area. Consequently, subsequent executions of a program that defines one or more applications will usually have the same ID returned for the application each time. The same is true when different programs define the same application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for application definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate application names to pass on the **arm_init** subroutine call.

### Files

| | |
|---|---|
| **/usr/include/arm.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library. |

### Related Information

“ARM Contexts in Spmi Data Space” on page 195.

## arm_getid Subroutine

### Purpose

The **arm_getid** subroutine is used to register a transaction as belonging to an application and assign a unique identifier to the application/transaction pair. In the PTX implementation of ARM, multiple instances of a transaction within one application can’t be defined. See “Implementation Specifics” on page 308. A transaction must be registered before any ARM measurements can begin.

## Library

ARM Library (**libarm.a**).

## Syntax

```
#include arm.h

arm_tran_id_t arm_getid(       arm_appl_id_t appl_id,       /* application handle
*/
      arm_ptr_t     *tran_name,    /* transaction name              */
      arm_ptr_t     *tran_detail,  /* transaction additional info   */
      arm_flag_t    flags,         /* Reserved = 0                  */
      arm_data_t    *data,         /* Reserved = NULL               */
      arm_data_sz_t data_size);    /* Reserved = 0                  */
```

## Description

Each transaction needs to be defined by a unique name within an application. Transactions can be defined so they best fit the application environment. For example, if a given environment has thousands of unique transactions, it may be feasible to define groups of similar transactions to prevent data overload. In other situations, you may want to use generated transaction names that reflect what data a transaction carries along with the transaction type. For example, the type of SQL query could be analyzed to group customer query transactions according to complexity, such as *customer_simple*, *customer*, *customer_complex*. Whichever method is used to name transactions, in the PTX implementation of the ARM API, measurements are always collected for each unique combination of:

1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

## Parameters

### appl_id

The identifier returned by an earlier call to **arm_init** ("arm_init Subroutine" on page 305). The PTX implementation does not require that the **arm_init** subroutine call was issued by the same program or process now issuing the **arm_getid** subroutine call. However, the number of issued **arm_init** subroutine calls for the application name must exceed the number of issued **arm_end** subroutine calls for this *appl_id*.

The *appl_id* is used to look for an application structure. If one is not found or if the use-count of the one found is zero, no action is taken and the function returns -1.

### tran_name

A unique transaction name. The name only needs to be unique within the *appl_id*. The maximum length is 128 characters including the terminating zero. The argument is converted to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This key is used to look for a transaction structure (that belongs to the application identified in the first argument) in the library's private shared memory area. If a transaction structure is found, its use-count is set to one and the transaction ID stored in the structure is returned to the caller. If the structure is not found, one is created and assigned the next free transaction ID, given a use-count of one and added to the application's linked list of transactions. The new assigned transaction ID is returned to the caller.

Up-to 64 bytes, including the terminating zero, of the *tran_name* parameter is saved as the description of the **SpmiCx context** ("SpmiCx Structure" on page 206) that represents the transaction in the Spmi hierarchy. The key is used as the short name of the context.

### tran_detail

Can be passed in as NULL or some means of specifying a unique instance of the transaction. In the PTX implementation of the ARM API, this parameter is ignored. Consequently, it is not

possible to define unique instances of a transaction. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes in length, including the terminating zero.

For the implementation to take this argument in use, another context level would have to be defined between the application context and the transaction context. This was deemed excessive.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

### Return Values

If successful, the subroutine returns an **tran_id** application identifier. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the **arm_start** ("arm_start Subroutine") subroutine, which will cause **arm_start** to function as a no-operation.

### Error Codes

No error codes are defined by the PTX implementation of the ARM API.

### Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the use-count for a transaction structure is either one or zero. This ensures that as long as the application structure is active, so are all transactions for which an **arm_getid** call was issued after the application was activated by an **arm_init** ("arm_init Subroutine" on page 305) call. The transaction use-count is reset to zero by the **arm_end** ("arm_end Subroutine" on page 313) call if this call causes the application use-count to go to zero.

Note that the implementation of **arm_getid** doesn't allow unique instances of a transaction to be defined. The **tran_id** associated with a transaction is stored in the ARM shared memory area and will remain constant throughout the life of the shared memory area. Consequently, subsequent executions of a program that defines one or more transactions under a given application will usually have the same ID returned for the transactions each time. The same is true when different programs define the same transaction within an application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for transaction definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate transaction names to pass on the **arm_getid** subroutine call.

### Files

| | |
|---|---|
| **/usr/include/arm.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library. |

### Related Information

"ARM Contexts in Spmi Data Space" on page 195, **arm_init** ("arm_init Subroutine" on page 305) subroutine, **arm_end** ("arm_end Subroutine" on page 313) subroutine.

## arm_start Subroutine

### Purpose

The **arm_start** subroutine is used to mark the beginning of the execution of a transaction. Measurement of the transaction response time starts at the execution of this subroutine.

## Library

ARM Library (**libarm.a**).

## Syntax

```
#include arm.h

arm_start_handle_t arm_start( arm_tran_id_t tran_id,   /* transaction name identifier
*/
       arm_flag_t    flags,         /* Reserved = 0                  */
       arm_data_t    *data,         /* Reserved = NULL               */
       arm_data_sz_t data_size);    /* Reserved = 0                  */
```

## Description

Each **arm_start** subroutine call marks the beginning of another instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held until the execution of a matching **arm_stop** ("arm_stop Subroutine" on page 311) subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:

1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

## Parameters

**tran_id**

The identifier is returned by an earlier call to **arm_getid**, "arm_getid Subroutine" on page 306. The PTX implementation does not require that the **arm_getid** subroutine call was issued by the same program or process now issuing the **arm_start** subroutine call. However, the transaction's application structure must be active, which means that the number of issued **arm_init** subroutine calls for the application name must exceed the number of issued **arm_end** subroutine calls for the application's ***appl_id***. If an application was inactivated by issuing a sufficient number of **arm_end** calls, all transactions defined for that application will have their use_count set to zero. The count remains zero (and the transaction inactive) until a new **arm_getid** subroutine is issued for the transaction.

The *tran_id* argument is used to look for a transaction structure. If one is not found or if the use-count of the one found is zero, no action is taken and the function returns -1. If one is found, a transaction instance structure (called a *slot structure*) is allocated, assigned the next free instance ID, and updated with the start time of the transaction instance. The assigned instance ID is returned to the caller.

In compliance with the ARM API specifications, if the *tran_id* passed is one returned from a previous **arm_getid** subroutine call that failed, the **arm_start** subroutine call functions as a no-operation function. It will return a NULL **start_handle**, which can be passed to subsequent **arm_update** ("arm_update Subroutine" on page 310) and **arm_stop** ("arm_stop Subroutine" on page 311) subroutine calls with the effect that those calls are no-operation functions.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

## Return Values

If successful, the subroutine returns a **start_handle**, which uniquely defines this transaction execution instance. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the **arm_update** ("arm_update Subroutine" on page 310) and **arm_stop** ("arm_stop Subroutine" on page 311) subroutines, which will cause those subroutines to operate as no-operation functions.

## Error Codes

No error codes are defined by the PTX implementation of the ARM API.

## Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

## Files

**/usr/include/arm.h** Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library.

## Related Information

"ARM Contexts in Spmi Data Space" on page 195, **arm_init** ("arm_init Subroutine" on page 305) subroutine, **arm_getid** ("arm_getid Subroutine" on page 306) subroutine, **arm_end** ("arm_end Subroutine" on page 313) subroutine.

# arm_update Subroutine

## Purpose

The **arm_update** subroutine is used to collect information about a transaction's progress. It is a no-operation subroutine in the PTX implementation.

## Library

ARM Library (**libarm.a**).

## Syntax

```
#include arm.h

arm_ret_stat_t arm_update(  arm_start_handle_t arm_handle, /* unique transaction handle
*/
        arm_flag_t        flags,      /* Reserved = 0              */
        arm_data_t        *data,      /* Reserved = NULL           */
        arm_data_sz_t     data_size); /* Reserved = 0              */
```

## Description

The **arm_update** subroutine is implemented as a no-operation in the PTX version of the ARM API. It is intended to be used for providing status information for a long-running transaction. Because there's no feasible way to display such information in current PTX monitors, the subroutine is a NULL function.

## Parameters

### start_handle

The identifier is returned by an earlier call to **arm_start**, "arm_start Subroutine" on page 308. The *start_handle* argument is used to look for the *slot structure* created by the **arm_start** subroutine call. If one is not found, no action is taken and the function returns -1. Otherwise a zero is returned.

In compliance with the ARM API specifications, if the *start_handle* passed is one returned from a previous **arm_start** subroutine call that failed, or from an **arm_start** subroutine operating as a no-operation function, the **arm_update** subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

### flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

## Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned.

## Error Codes

No error codes are defined by the PTX implementation of the ARM API.

## Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product. It is implemented as a NULL subroutine call.

## Files

| | |
|---|---|
| **/usr/include/arm.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library. |

## Related Information

"ARM Contexts in Spmi Data Space" on page 195, **arm_init** ("arm_init Subroutine" on page 305) subroutine, **arm_getid** ("arm_getid Subroutine" on page 306) subroutine, **arm_start** ("arm_start Subroutine" on page 308) subroutine, **arm_stop** ("arm_stop Subroutine") subroutine, **arm_end** ("arm_end Subroutine" on page 313) subroutine.

# arm_stop Subroutine

## Purpose

The **arm_stop** subroutine is used to mark the end of the execution of a transaction. Measurement of the transaction response time completes at the execution of this subroutine.

## Library

ARM Library (**libarm.a**).

## Syntax

```
#include arm.h

arm_ret_stat_t arm_stop( arm_start_handle_t arm_handle,  /* unique transaction handle
*/
       const arm_status_t comp_status, /* Good=0, Abort=1, Failed=2  */
       arm_flag_t         flags,       /* Reserved = 0               */
       arm_data_t         *data,       /* Reserved = NULL            */
       arm_data_sz_t      data_size);  /* Reserved = 0               */
```

## Description

Each **arm_stop** subroutine call marks the end of an instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held from the execution of the **arm_start** ("arm_start Subroutine" on page 308) subroutine call and until the execution of a matching **arm_stop** subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:

1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

## Parameters

### arm_handle

The identifier is returned by an earlier call to **arm_start**, "arm_start Subroutine" on page 308. The *arm_handle* argument is used to look for a *slot structure* created by the **arm_start** ("arm_start Subroutine" on page 308) call, which returned this *arm_handle*. If one is not found, no action is

taken and the function returns -1. If one is found, a *post structure* is allocated and added to the linked list of post structures used to pass data to the **SpmiArmd** daemon. The post structure is updated with the start time from the slot structure, the path to the transaction context, and the stop time of the transaction instance.

In compliance with the ARM API specifications, if the *start_handle* passed is one returned from a previous **arm_start** subroutine call that failed, or from an **arm_start** subroutine operating as a no-operation function, the **arm_stop** subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

**comp_status**

User supplied transaction completion code. The following codes are defined:

- **ARM_GOOD** - successful completion
  Response time is calculated. The response time is calculated as a fixed point value in milliseconds and saved in the metric **resptime** (page "ARM Transaction Metrics" on page 196). In addition, the weighted average response time (in **respavg** (page "ARM Transaction Metrics" on page 196) is calculated as a floating point value using a variable *weight* (page "ARM Transaction Metrics" on page 196) that defaults to 75%. The average response time is calculated as *weight* percent of the previous value of the average plus (100 - *weight*) percent of the latest response time observation. The value of *weight* can be changed from the **SpmiArmd** daemon's configuration file **/etc/perf/SpmiArmd.cf**. In addition, the maximum and minimum response time for this transaction is updated, if required. Finally the **count** (page "ARM Transaction Metrics" on page 196) of successful transaction executions is incremented.
- **ARM_ABORT** - transaction aborted
  The **aborted** (page "ARM Transaction Metrics" on page 196) counter is incremented. No other updates occur.
- **ARM_FAILED** - transaction failed
  The **failed** (page "ARM Transaction Metrics" on page 196) counter is incremented. No other updates occur.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

## Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned.

## Error Codes

No error codes are defined by the PTX implementation of the ARM API.

## Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/arm.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library. |

## Related Information

"ARM Contexts in Spmi Data Space" on page 195, **arm_init** ("arm_init Subroutine" on page 305) subroutine, **arm_getid** ("arm_getid Subroutine" on page 306) subroutine, **arm_start** ("arm_start Subroutine" on page 308) subroutine, **arm_end** ("arm_end Subroutine" on page 313) subroutine.

# arm_end Subroutine

## Purpose

The **arm_end** subroutine is used to mark the end of an application. This subroutine call must always be called when a program that issued an **arm_init** (“arm_init Subroutine” on page 305) subroutine call terminates. In the PTX implementation of ARM, application data structures may persist after **arm_end** is issued. See “Implementation Specifics” on page 314.

## Library

ARM Library (**libarm.a**).

## Syntax

```
#include arm.h

arm_ret_stat_t ARM_API arm_end(  arm_appl_id_t appl_id,       /* application id
*/
       arm_flag_t    flags,         /* Reserved = 0           */
       arm_data_t    *data,         /* Reserved = NULL        */
       arm_data_sz_t data_size);    /* Reserved = 0           */
```

## Description

By calling the **arm_end** subroutine, an application program signals to the ARM library that it has ceased issuing ARM subroutine calls for the application specified and that the library code can remove references to the application. As far as the calling program is concerned, all references to transactions defined for the named application can be removed as well.

## Parameters

### appl_id

The identifier is returned by an earlier call to **arm_init**, “arm_init Subroutine” on page 305. The PTX implementation does not require that the **arm_init** subroutine call was issued by the same program or process now issuing the **arm_end** subroutine call. However, each time the **arm_end** subroutine call is issued against an *appl_id*, the use-count of the transaction structure is decremented. When the count reaches zero, the application structure and all associated transaction structures are marked as inactive. Subsequent **arm_init** calls can reactivate the application structure but transaction structures formerly associated with the application are not automatically activated. Each transaction must be reactivated through the **arm_getid** (“arm_getid Subroutine” on page 306) subroutine call.

The *appl_id* is used to look for an application structure. If none is found, no action is taken and the function returns -1. If one is found, the use-count of the application structure is decremented. If that makes the counter zero, the use-counts of all associated transaction structures are set to zero. The total number of application structures that have been initialized for the calling process but not ended is decremented. If this count reaches zero, access to the shared memory from the process is released and the count of users of the shared memory area is decremented. If the count of users of the shared memory segment reaches zero, the shared memory segment is deleted.

### flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

## Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned.

## Error Codes

No error codes are defined by the PTX implementation of the ARM API.

### Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that, in the PTX implementation of ARM, multiple processes can issue **arm_init** ("arm_init Subroutine" on page 305) subroutine calls for a given application with the effect that multiple simultaneous definitions of the application are effective. The ARM library code points all these definitions to a single application structure in the ARM private shared memory area. A use-count keeps track of the number of simultaneous definitions. Each time **arm_init** is issued for the application name, the counter is incremented and each time the **arm_end** subroutine call is issued for the associated *appl_id*, the counter is decremented. No call to **arm_end** is permitted to decrement the counter less than zero.

Only when the counter reaches zero is the application structure inactivated. As long as the counter is non-zero, transactions defined for the application remain active and new transactions can be defined for the application. It does not matter which process created the definition of the application.

This implementation was chosen because it makes perfect sense in a PTX environment. Any more restrictive implementation would have increased memory use significantly and would be useless for PTX monitoring purposes.

### Files

| | |
|---|---|
| **/usr/include/arm.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library. |

### Related Information

"ARM Contexts in Spmi Data Space" on page 195, **arm_init** ("arm_init Subroutine" on page 305) subroutine, **arm_getid** ("arm_getid Subroutine" on page 306) subroutine.

## ARM Replacement Library Implementation

This section describes the implementation of the PTX version of the ARM replacement library. The replacement library differs from the standard PTX ARM implementation by invoking an earlier installed ARM implementation in addition to invoking the PTX implementation. For the following description, the previously installed ARM library is referred to as the *lower library*.

## arm_init Dual Call Subroutine

### Purpose

The **arm_init** subroutine is used to define an application or a unique instance of an application to the ARM library. While, in the PTX implementation of ARM, instances of applications can't be defined, the ARM implementation in the *lower library* may permit this. An application must be defined before any other ARM subroutine is issued.

### Library

ARM Library (**libarm.a**).

### Syntax

```
#include arm.h

arm_appl_id_t arm_init(  arm_ptr_t     *appname,      /* application name
*/
      arm_ptr_t     *appl_user_id, /* Name of the application user  */
      arm_flag_t    flags,         /* Reserved = 0                  */
      arm_data_t    *data,         /* Reserved = NULL               */
      arm_data_sz_t data_size);    /* Reserved = 0                  */
```

## Description

Each application needs to be defined by a unique name. An application can be defined as loosely or as rigidly as required. It may be defined as a single execution of one program, multiple (possibly simultaneous) executions of one program, or multiple executions of multiple programs that together constitute an application. Any one user of ARM may define the application so it best fits the measurement granularity desired. For the PTX implementation, measurements are always collected for each unique combination of:

1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

Before the PTX implementation code is executed, the *lower library* is called. If this call returns a value greater than zero, that return value is passed to the caller as the application ID. If the returned value from the *lower library* is zero or negative, the return value is the one generated by the PTX library code.

## Parameters

**appname**

A unique application name. The maximum length is 128 characters including the terminating zero. The PTX library code converts this value to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This key is used to look for an application structure in the library's private shared memory area. If a structure is found, its use-count is incremented and the application ID stored in the structure is saved. If the structure is not found, one is created, assigned the next free application ID and given a use-count of one. The new assigned application ID is saved. If the call to the *lower library* was successful, a cross-reference is created from the *lower library's* application ID to the PTX library's application ID for use by **arm_getid** (“arm_getid Dual Call Subroutine” on page 316) and **arm_end** (“arm_end Dual Call Subroutine” on page 323).

Up-to 64 bytes, including the terminating zero, of the *appname* parameter is saved as the description of the **SpmiCx context** (“SpmiCx Structure” on page 206) that represents the application in the Spmi hierarchy. The key is used as the short name of the context.

**appl_user_id**

Can be passed in as NULL or some means of specifying a user ID for the application. This allows the calling program to define unique instances of an application. In the PTX implementation of the ARM API, this parameter is ignored. Consequently, it is not possible to define unique instances of an application. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes in length, including the terminating zero.

For the PTX implementation to take this argument in use, another context level would have to be defined between the application context and the transaction context. This was deemed excessive.

For the *lower library* implementation of this subroutine call, the *appl_user_id* argument may have significance. If so, it's transparent to the PTX implementation.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

## Return Values

If the call to the *lower library* was successful, the subroutine returns an **appl_id** application identifier as returned from the *lower library*. If the subroutine call to the *lower library* fails but the PTX implementation doesn't fail, the **appl_id** returned is the one assigned by the PTX library. If both implementations fail, a value less than zero is returned.

## Error Codes

No error codes are defined by the PTX implementation of the ARM API.

## Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the implementation of **arm_init** doesn't allow unique instances of an application to be defined. The **appl_id** associated with an application is stored in the ARM shared memory area and will remain constant throughout the life of the shared memory area. Consequently, subsequent executions of a program that defines one or more applications will usually have the same ID returned for the application each time. The same is true when different programs define the same application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for application definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate application names to pass on the **arm_init** subroutine call.

Regardless of the implementation restrictions of the PTX library, the *lower library* may or may not have its own implementation restrictions.

## Files

| | |
|---|---|
| **/usr/include/arm.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library. |

## Related Information

"ARM Contexts in Spmi Data Space" on page 195.

# arm_getid Dual Call Subroutine

## Purpose

The **arm_getid** subroutine is used to register a transaction as belonging to an application and assign a unique identifier to the application/transaction pair. In the PTX implementation of ARM, multiple instances of a transaction within one application can't be defined. The *lower library* implementation of this subroutine may provide support for instances of transactions. A transaction must be registered before any ARM measurements can begin.

## Library

ARM Library (**libarm.a**).

## Syntax

```
#include arm.h

arm_tran_id_t arm_getid(       arm_appl_id_t appl_id,       /* application handle
*/
      arm_ptr_t     *tran_name,    /* transaction name              */
      arm_ptr_t     *tran_detail,  /* transaction additional info   */
      arm_flag_t    flags,         /* Reserved = 0                  */
      arm_data_t    *data,         /* Reserved = NULL               */
      arm_data_sz_t data_size);    /* Reserved = 0                  */
```

## Description

Each transaction needs to be defined by a unique name within an application. Transactions can be defined so they best fit the application environment. For example, if a given environment has thousands of unique transactions, it may be feasible to define groups of similar transactions to prevent data overload. In other situations, you may want to use generated transaction names that reflect what data a transaction carries along with the transaction type. For example, the type of SQL query could be analyzed to group customer query transactions according to complexity, such as *customer_simple*, *customer*, *customer_complex*.

Whichever method is used to name transactions, in the PTX implementation of the ARM API, measurements are always collected for each unique combination of:

1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

Before the PTX implementation code is executed, the *lower library* is called. If this call returns a value greater than zero, that return value is passed to the caller as the transaction ID. If the returned value from the *lower library* is zero or negative, the return value is the one generated by the PTX library code.

## Parameters

### appl_id

The identifier returned by an earlier call to **arm_init** ("arm_init Dual Call Subroutine" on page 314). The identifier is passed to the **arm_getid** function of the *lower library*. If the *lower library* returns an identifier greater than zero, that identifier is the one that'll eventually be returned to the caller. After the invocation of the *lower library*, the PTX implementation attempts to translate the *appl_id* argument to its own identifier by consulting the cross-reference table created by **arm_init**. If one can be found, it is used for the PTX implementation; if no cross reference is found, the *appl_id* is used as passed in. The PTX implementation does not require that the **arm_init** subroutine call was issued by the same program or process now issuing the **arm_getid** subroutine call. However, the number of issued **arm_init** subroutine calls for the application name must exceed the number of issued **arm_end** subroutine calls for this *appl_id*.

In the PTX implementation, the *appl_id* (as retrieved from the cross-reference table) is used to look for an application structure. If one is not found or if the use-count of the one found is zero, the PTX implementation is considered to have failed and no action is taken by the PTX library.

### tran_name

A unique transaction name. The name only needs to be unique within the *appl_id*. The maximum length is 128 characters including the terminating zero. In the PTX implementation, the argument is converted to a key by removing all blanks and truncating the string to 32 characters, including a terminating zero. This key is used to look for a transaction structure (that belongs to the application identified in the first argument) in the library's private shared memory area. If a transaction structure is found, its use-count is set to one and the transaction ID stored in the structure is saved. If the structure is not found, one is created and assigned the next free transaction ID, given a use-count of one and added to the application's linked list of transactions. The new assigned transaction ID is saved. If the call to the *lower library* was successful, a cross-reference is created from the *lower library's* transaction ID to the PTX library's transaction ID for use by **arm_start** ("arm_start Dual Call Subroutine" on page 319).

Up-to 64 bytes, including the terminating zero, of the *tran_name* parameter is saved as the description of the **SpmiCx context** ("SpmiCx Structure" on page 206) that represents the transaction in the Spmi hierarchy. The key is used as the short name of the context.

### tran_detail

Can be passed in as NULL or some means of specifying a unique instance of the transaction. In the PTX implementation of the ARM API, this parameter is ignored. Consequently, it is not possible to define unique instances of a transaction. If specified as non-NULL, this parameter must be a string not exceeding 128 bytes in length, including the terminating zero.

For the implementation to take this argument in use, another context level would have to be defined between the application context and the transaction context. This was deemed excessive.

For the *lower library* implementation of this subroutine call, the *tran_detail* argument may have significance. If so, it's transparent to the PTX implementation.

### flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

## Return Values

If successful, the subroutine returns an **tran_id** application identifier. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the **arm_start** (“arm_start Dual Call Subroutine” on page 319) subroutine, which will cause **arm_start** to function as a no-operation.

If the call to the *lower library* was successful, the **tran_id** transaction identifier returned is the one assigned by the *lower library*. If the subroutine call to the *lower library* failed but the PTX implementation didn’t fail, the **tran_id** returned is the one assigned by the PTX library. If both implementations fail, a value less than zero is returned. In compliance with the ARM API specification, an error return value can be passed to the **arm_start** (“arm_start Dual Call Subroutine” on page 319) subroutine, which will cause **arm_start** to function as a no-operation.

## Error Codes

No error codes are defined by the PTX implementation of the ARM API.

## Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that the use-count for a transaction structure is either one or zero. This ensures that as long as the application structure is active, so are all transactions for which an **arm_getid** call was issued after the application was activated by an **arm_init** (“arm_init Dual Call Subroutine” on page 314) call. The transaction use-count is reset to zero by the **arm_end** (“arm_end Dual Call Subroutine” on page 323) call if this call causes the application use-count to go to zero.

Note that the implementation of **arm_getid** doesn’t allow unique instances of a transaction to be defined. The **tran_id** associated with a transaction is stored in the ARM shared memory area and will remain constant throughout the life of the shared memory area. Consequently, subsequent executions of a program that defines one or more transactions under a given application will usually have the same ID returned for the transactions each time. The same is true when different programs define the same transaction within an application: As long as the shared memory area exists, they will all have the same ID returned. This is done to minimize the use of memory for transaction definitions and because it makes no difference from a PTX point of view.

If this is not acceptable from an application point of view, programs can dynamically generate transaction names to pass on the **arm_getid** subroutine call.

Regardless of the implementation restrictions of the PTX library, the *lower library* may or may not have its own implementation restrictions.

## Files

| | |
|---|---|
| **/usr/include/arm.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library. |

## Related Information

“ARM Contexts in Spmi Data Space” on page 195, **arm_init** (“arm_init Dual Call Subroutine” on page 314) subroutine, **arm_end** (“arm_end Dual Call Subroutine” on page 323) subroutine.

# arm_start Dual Call Subroutine

## Purpose

The **arm_start** subroutine is used to mark the beginning of the execution of a transaction. Measurement of the transaction response time starts at the execution of this subroutine.

## Library

ARM Library (**libarm.a**).

## Syntax

```
#include arm.h

arm_start_handle_t arm_start( arm_tran_id_t tran_id,     /* transaction name identifier
*/
       arm_flag_t    flags,         /* Reserved = 0                   */
       arm_data_t    *data,         /* Reserved = NULL                */
       arm_data_sz_t data_size);    /* Reserved = 0                   */
```

## Description

Each **arm_start** subroutine call marks the beginning of another instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held until the execution of a matching **arm_stop** ("arm_stop Dual Call Subroutine" on page 322) subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:

1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

Before the PTX implementation code is executed, the *lower library* is called. If this call returns a value greater than zero, that return value is passed to the caller as the start handle. If the value returned by the *lower library* is zero or negative, the return value is the one generated by the PTX library code.

## Parameters

**tran_id**

The identifier is returned by an earlier call to **arm_getid**, "arm_getid Dual Call Subroutine" on page 316. The identifier is passed to the **arm_start** function of the *lower library*. If the *lower library* returns an identifier greater than zero, that identifier is the one that'll eventually be returned to the caller. After the invocation of the *lower library*, the PTX implementation attempts to translate the *tran_id* argument to its own identifier from the cross-reference table created by **arm_getid**. If one can be found, it is used for the PTX implementation; if no cross reference is found, the *tran_id*is used as passed in.The PTX implementation does not require that the **arm_getid** subroutine call was issued by the same program or process now issuing the **arm_start** subroutine call. However, the transaction's application structure must be active, which means that the number of issued **arm_init** subroutine calls for the application name must exceed the number of issued **arm_end** subroutine calls for the application's ***appl_id***. If an application was inactivated by issuing a sufficient number of **arm_end** calls, all transactions defined for that application will have their use_count set to zero. The count remains zero (and the transaction inactive) until a new **arm_getid** subroutine is issued for the transaction.

In the PTX implementation, the *tran_id* (as retrieved from the cross-reference table) is used to look for a transaction structure. If one is not found or if the use-count of the one found is zero, the PTX implementation is considered to have failed and no action is taken by the PTX library. If one is found, a transaction instance structure (called a *slot structure*) is allocated, assigned the next free instance ID, and updated with the start time of the transaction instance. The assigned instance ID is saved as the **start_handle**. If the call to the *lower library* was successful, a cross-reference is

created from the *lower library's* start_handle to the PTX library's start_handle for use by **arm_update** ("arm_update Dual Call Subroutine") and **arm_stop** ("arm_stop Dual Call Subroutine" on page 322).

In compliance with the ARM API specifications, if the *tran_id* passed is one returned from a previous **arm_getid** subroutine call that failed, the **arm_start** subroutine call functions as a no-operation function. It will return a NULL **start_handle**, which can be passed to subsequent **arm_update** ("arm_update Dual Call Subroutine") and **arm_stop** ("arm_stop Dual Call Subroutine" on page 322) subroutine calls with the effect that those calls are no-operation functions.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

### Return Values

If successful, the subroutine returns a **start_handle**, which uniquely defines this transaction execution instance. If the subroutine fails, a value less than zero is returned. In compliance with the ARM API specifications, the error return value can be passed to the **arm_update** ("arm_update Dual Call Subroutine") and **arm_stop** ("arm_stop Dual Call Subroutine" on page 322) subroutines, which will cause those subroutines to operate as no-operation functions.

If the call to the *lower library* was successful, the **start_handle** instance ID returned is the one assigned by the *lower library*. If the subroutine call to the *lower library* failed but the PTX implementation didn't fail, the **start_handle** returned is the one assigned by the PTX library. If both implementations fail, a value less than zero is returned.

### Error Codes

No error codes are defined by the PTX implementation of the ARM API.

### Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

### Files

| | |
|---|---|
| **/usr/include/arm.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library. |

### Related Information

"ARM Contexts in Spmi Data Space" on page 195, **arm_init** ("arm_init Dual Call Subroutine" on page 314) subroutine, **arm_getid** ("arm_getid Dual Call Subroutine" on page 316) subroutine, **arm_end** ("arm_end Dual Call Subroutine" on page 323) subroutine.

## arm_update Dual Call Subroutine

### Purpose

The **arm_update** subroutine is used to collect information about a transaction's progress. It is a no-operation subroutine in the PTX implementation but may be fully implemented by the *lower library*.

### Library

ARM Library (**libarm.a**).

### Syntax

```
#include arm.h

arm_ret_stat_t arm_update(  arm_start_handle_t arm_handle, /* unique transaction handle
```

```
*/
        arm_flag_t         flags,      /* Reserved = 0                */
        arm_data_t         *data,      /* Reserved = NULL             */
        arm_data_sz_t      data_size); /* Reserved = 0                */
```

## Description

The **arm_update** subroutine is implemented as a no-operation in the PTX version of the ARM API. It is intended to be used for providing status information for a long-running transaction. Because there's no feasible way to display such information in current PTX monitors, the subroutine is a NULL function.

The *lower library* implementation of the **arm_update** subroutine is always invoked.

## Parameters

**start_handle**

The identifier is returned by an earlier call to **arm_start**, "arm_start Dual Call Subroutine" on page 319. The identifier is passed to the **arm_update** function of the *lower library*. If the *lower library* returns a zero return code., that return code is returned to the caller. After the invocation of the *lower library*, the PTX implementation attempts to translate the *arm_handle*argument to its own identifier from the cross-reference table created by **arm_start**. If one can be found, it is used for the PTX implementation; if no cross reference is found, the *arm_handle* is used as passed in. The PTX implementation uses the *start_handle* argument to look for the *slot structure* created by the **arm_start** subroutine call. If one is found the PTX implementation is considered to have succeeded, otherwise it is considered to have failed.

In compliance with the ARM API specifications, if the *start_handle* passed is one returned from a previous **arm_start** subroutine call that failed, or from an **arm_start** subroutine operating as a no-operation function, the **arm_update** subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

## Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned. If the call to the *lower library* was successful, a zero is returned. If the subroutine call to the *lower library* failed but the PTX implementation didn't fail, a zero is returned. If both implementations failed, a value less than zero is returned.

## Error Codes

No error codes are defined by the PTX implementation of the ARM API.

## Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product. It is implemented as a NULL subroutine call.

## Files

| | |
|---|---|
| **/usr/include/arm.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library. |

## Related Information

"ARM Contexts in Spmi Data Space" on page 195, **arm_init** ("arm_init Dual Call Subroutine" on page 314) subroutine, **arm_getid** ("arm_getid Dual Call Subroutine" on page 316) subroutine, **arm_start** ("arm_start

Dual Call Subroutine" on page 319) subroutine, **arm_stop** ("arm_stop Dual Call Subroutine") subroutine, **arm_end** ("arm_end Dual Call Subroutine" on page 323) subroutine.

## arm_stop Dual Call Subroutine

### Purpose

The **arm_stop** subroutine is used to mark the end of the execution of a transaction. Measurement of the transaction response time completes at the execution of this subroutine.

### Library

ARM Library (**libarm.a**).

### Syntax

```
#include arm.h

arm_ret_stat_t arm_stop(  arm_start_handle_t arm_handle,  /* unique transaction handle
*/
       const arm_status_t comp_status, /* Good=0, Abort=1, Failed=2  */
       arm_flag_t         flags,       /* Reserved = 0               */
       arm_data_t         *data,       /* Reserved = NULL            */
       arm_data_sz_t      data_size);  /* Reserved = 0               */
```

### Description

Each **arm_stop** subroutine call marks the end of an instance of a transaction within an application. Multiple instances (simultaneous executions of the transaction) may exist. Control information for the transaction instance is held from the execution of the **arm_start** ("arm_start Dual Call Subroutine" on page 319) subroutine call and until the execution of a matching **arm_stop** subroutine call, at which time the elapsed time is calculated and used to update transaction measurement metrics for the transaction. Metrics are accumulated for each unique combination of the following three components:

1. Hostname of the machine where the instrumented application executes.
2. Unique application name.
3. Unique transaction name.

Before the PTX implementation code is executed, the *lower library* is called. If this call returns a value of zero, that return value is passed to the caller. If the value returned by the *lower library* is non-zero, the return value is the one generated by the PTX library code.

### Parameters

**arm_handle**

The identifier is returned by an earlier call to **arm_start**, "arm_start Dual Call Subroutine" on page 319. The identifier is passed to the **arm_stop** function of the *lower library*. If the *lower library* returns a zero return code, that return code is returned to the caller. After the invocation of the *lower library*, the PTX implementation attempts to translate the *arm_handle*argument to its own identifier from the cross-reference table created by **arm_start**. If one can be found, it is used for the PTX implementation; if no cross reference is found, the *arm_handle* is used as passed in. The PTX implementation uses the *start_handle* argument to look for the *slot structure* created by the **arm_start** subroutine call. If one is found, a *post structure* is allocated and added to the linked list of post structures used to pass data to the **SpmiArmd** daemon. The post structure is updated with the start time from the slot structure, the path to the transaction context, and the stop time of the transaction instance. If no *slot structure* was found, the PTX implementation is considered to have failed.

In compliance with the ARM API specifications, if the *start_handle* passed is one returned from a previous **arm_start** subroutine call that failed, or from an **arm_start** subroutine operating as a no-operation function, the **arm_stop** subroutine call executes as a no-operation function. It will return a zero to indicate successful completion.

**comp_status**

User supplied transaction completion code. The following codes are defined:

- **ARM_GOOD** - successful completion
  Response time is calculated. The response time is calculated as a fixed point value in milliseconds and saved in the metric **resptime** (page “ARM Transaction Metrics” on page 196). In addition, the weighted average response time (in **respavg** (page “ARM Transaction Metrics” on page 196)) is calculated as a floating point value using a variable *weight* (page “ARM Transaction Metrics” on page 196), that defaults to 75%. The average response time is calculated as *weight* percent of the previous value of the average plus (100 - *weight*) percent of the latest response time observation. The value of *weight* can be changed from the **SpmiArmd** daemon’s configuration file **/etc/perf/SpmiArmd.cf**. In addition, the maximum and minimum response time for this transaction is updated, if required. Finally the **count** (page “ARM Transaction Metrics” on page 196) of successful transaction executions is incremented.
- **ARM_ABORT** - transaction aborted
  The **aborted** (page “ARM Transaction Metrics” on page 196) counter is incremented. No other updates occur.
- **ARM_FAILED** - transaction failed
  The **failed** (page “ARM Transaction Metrics” on page 196) counter is incremented. No other updates occur.

**flags, data, data_size**

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

## Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned. If the call to the *lower library* was successful, a zero is returned. If the subroutine call to the *lower library* failed but the PTX implementation didn’t fail, a zero is returned. If both implementations failed, a value less than zero is returned.

## Error Codes

No error codes are defined by the PTX implementation of the ARM API.

## Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/arm.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library. |

## Related Information

“ARM Contexts in Spmi Data Space” on page 195, **arm_init** (“arm_init Dual Call Subroutine” on page 314) subroutine, **arm_getid** (“arm_getid Dual Call Subroutine” on page 316) subroutine, **arm_start** (“arm_start Dual Call Subroutine” on page 319) subroutine, **arm_end** (“arm_end Dual Call Subroutine”) subroutine.

# arm_end Dual Call Subroutine

## Purpose

The **arm_end** subroutine is used to mark the end of an application. This subroutine call must always be called when a program that issued an **arm_init** (“arm_init Dual Call Subroutine” on page 314) subroutine

call terminates. In the PTX implementation of ARM, application data structures may persist after **arm_end** is issued. See "Implementation Specifics" on page 314. This may not be the case for the *lower library* implementation.

## Library

ARM Library (**libarm.a**).

## Syntax

```
#include arm.h

arm_ret_stat_t ARM_API arm_end(  arm_appl_id_t appl_id,       /* application id
*/
       arm_flag_t    flags,         /* Reserved = 0                  */
       arm_data_t    *data,         /* Reserved = NULL               */
       arm_data_sz_t data_size);    /* Reserved = 0                  */
```

## Description

By calling the **arm_end** subroutine, an application program signals to the ARM library that it has ceased issuing ARM subroutine calls for the application specified and that the library code can remove references to the application. As far as the calling program is concerned, all references to transactions defined for the named application can be removed as well.

Before the PTX implementation code is executed, the *lower library* is called. If this call returns a value of zero, that return value is passed to the caller. If the value returned by the *lower library* is non-zero, the return value is the one generated by the PTX library code.

## Parameters

### appl_id

The identifier returned by an earlier call to **arm_init** ("arm_init Dual Call Subroutine" on page 314). The identifier is passed to the **arm_end** function of the *lower library*. If the *lower library* returns a zero, a zero is returned to the caller. After the invocation of the *lower library*, the PTX implementation attempts to translate the *appl_id* argument to its own identifier from the cross-reference table created by **arm_init** ("arm_init Dual Call Subroutine" on page 314). If one can be found, it is used for the PTX implementation; if no cross reference is found, the *appl_id* is used as passed in. The PTX implementation does not require that the **arm_init** subroutine call was issued by the same program or process now issuing the **arm_end** subroutine call. However, each time the **arm_end** subroutine call is issued against an *appl_id*, the use-count of the transaction structure is decremented. When the count reaches zero, the application structure and all associated transaction structures are marked as inactive. Subsequent **arm_init** calls can reactivate the application structure but transaction structures formerly associated with the application are not automatically activated. Each transaction must be reactivated through the **arm_getid** ("arm_getid Dual Call Subroutine" on page 316) subroutine call.

In the PTX implementation, the *appl_id* (as retrieved from the cross-reference table) is used to look for an application structure. If none is found, no action is taken and the PTX function is considered to have failed. If one is found, the use-count of the application structure is decremented. If that makes the counter zero, the use-counts of all associated transaction structures are set to zero. The total number of application structures that have been initialized for the calling process but not ended is decremented. If this count reaches zero, access to the shared memory from the process is released and the count of users of the shared memory area is decremented. If the count of users of the shared memory segment reaches zero, the shared memory segment is deleted.

### flags, data, data_size

In the current API definition, the last three arguments are for future use and they are ignored in the implementation.

## Return Values

If successful, the subroutine returns zero. If the subroutine fails, a value less than zero is returned. If the call to the *lower library* was successful, a zero is returned. If the subroutine call to the *lower library* failed but the PTX implementation didn't fail, a zero is returned. If both implementations failed, a value less than zero is returned.

## Error Codes

No error codes are defined by the PTX implementation of the ARM API.

## Implementation Specifics

This subroutine is part of the implementation of the ARM API in the Performance Toolbox for AIX licensed product.

Note that, in the PTX implementation of ARM, multiple processes can issue **arm_init** ("arm_init Dual Call Subroutine" on page 314) subroutine calls for a given application with the effect that multiple simultaneous definitions of the application are effective. The ARM library code points all these definitions to a single application structure in the ARM private shared memory area. A use-count keeps track of the number of simultaneous definitions. Each time **arm_init** is issued for the application name, the counter is incremented and each time the **arm_end** subroutine call is issued for the associated *appl_id*, the counter is decremented. No call to **arm_end** is permitted to decrement the counter less than zero.

Only when the counter reaches zero is the application structure inactivated. As long as the counter is non-zero, transactions defined for the application remain active and new transactions can be defined for the application. It does not matter which process created the definition of the application.

This implementation was chosen because it makes perfect sense in a PTX environment. Any more restrictive implementation would have increased memory use significantly and would be useless for PTX monitoring purposes.

For the implementation of **arm_end** in the *lower library*, other restrictions may exist.

## Files

| | |
|---|---|
| **/usr/include/arm.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the ARM library. |

## Related Information


- "ARM Contexts in Spmi Data Space" on page 195
- "arm_init Dual Call Subroutine" on page 314
- "arm_getid Dual Call Subroutine" on page 316

# Appendix E. SPMI Subroutines

The SPMI subroutines constitute the application programming interface (API) to the SPMI.

## SpmiAddSetHot Subroutine

### Purpose

Adds a set of peer statistics values to a hotset.

### Library

SPMI Library (**libSpmi.a**)

### Syntax

```
#include sys/Spmidef.h

struct SpmiHotVals *SpmiAddSetHot(HotSet, StatName,
GrandParent, maxresp,
                                  threshold, frequency, feed_type,
                                  except_type, severity, trap_no)
struct SpmiHotSet *HotSet;
char *StatName;
SpmiCxHdl GrandParent;
int maxresp;
int threshold;
int frequency;
int feed_type;
int excp_type;
int severity;
int trap_no;
```

### Description

The **SpmiAddSetHot** subroutine adds a set of peer statistics to a hotset. The **SpmiHotSet** (“SpmiHotSet Structure” on page 210) structure that provides the anchor point to the set must exist before the **SpmiAddSetHot** subroutine call can succeed.

### Parameters

**HotSet**
: Specifies a pointer to a valid structure of type **SpmiHotSet** as created by the **SpmiCreateHotSet** (“SpmiCreateHotSet” on page 330) subroutine call.

**StatName**
: Specifies the name of the statistic within the subcontexts (peer contexts) of the context identified by the *GrandParent* parameter.

**GrandParent**
: Specifies a valid **SpmiCxHdl** (“SpmiCxHdl Handle” on page 206) handle as obtained by another subroutine call. The handle must identify a context with at least one subcontext, which contains the statistic identified by the *StatName* parameter. If the context specified is one of the **RTime** contexts, no subcontext need to exist at the time the **SpmiAddSetHot** subroutine call is issued; the presence of the metric identified by the *StatName* parameter is checked against the context class description.

  If the context specified has or may have multiple levels of instantiable context below it (such as the **FS** and **RTime/ARM** contexts), the metric is only searched for at the lowest context level. The **SpmiHotSet** created is a pseudo hotvals structure used to link together a peer group of

**SpmiHotVals**structures, which are created under the covers, one for each subcontext of the *GrandParent* context. In the case of **RTime/ARM**, if additional contexts are later added under the *GrandParent* contexts, additional hotsets are added to the peer group. This is transparent to the application program, except that the **SpmiFirstHot**, **SpmiNextHot**, and **SpmiNextHotItem** subroutine calls will return the peer group **SpmiHotVals** pointer rather than the pointer to the pseudo structure.

Note that specifying a specific volume group context (such as **FS/rootvg**) or a specific application context (such as **RTime/ARN/armpeek**) is still valid and won't involve creation of pseudo **SpmiHotVals** structures.

**maxresp**

Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. If specified as zero, indicates that all "SpmiHotItems" on page 211 that meet the criteria specified by *threshold* must be returned, up-to a maximum of *maxresp* items. If both exceptions/traps and feeds are requested, the *maxresp* value is used to cap the number of exceptions/alerts as well as the number of items returned. If *feed_type* is specified as **SiHotAlways**, the *maxresp* parameter is still used to return at most *maxresp* items.

Where the *GrandParent* argument specifies a context that has multiple levels of instantiable contexts below it, the *maxresp* is applied to each of the lowest level contexts above the the actual peer contexts at a time. For example, if the *GrandParent* context is **FS** (file systems) and the system has three volume groups, then a *maxresp* value of 2 could cause up to a maximum of 2 x 3 = 6 responses to be generated.

**threshold**

Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. If specified as zero, indicates that all values read qualify to be returned in feeds. The value specified is compared to the data value read for each peer statistic. If the data value exceeds the *threshold*, it qualifies to be returned as an **SpmiHotItems** element in the **SpmiHotVals** ("SpmiHotVals Structure" on page 210) structure. If the *threshold* is specified as a negative value, the value qualifies if it is lower than the numeric value of *threshold*. If *feed_type* is specified as **SiHotAlways**, the threshold value is ignored for feeds. For peer statistics of type **SiCounter**, the *threshold* must be specified as a rate per second; for **SiQuantity** statistics the *threshold* is specified as a level.

**frequency**

Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. Ignored for feeds. Specifies the minimum number of minutes that must expire between any two exceptions/traps generated from this **SpmiHotVals** structure. This value must be specified as no less than 5 minutes.

**feed_type**

Specifies if feeds of **SpmiHotItems** should be returned for this **SpmiHotVals** structure. The following values are valid:

**SiHotNoFeed**
No feeds should be generated

**SiHotThreshold**
Feeds are controlled by *threshold*.

**SiHotAlways**
All values, up-to a maximum of *maxresp* must be returned as feeds.

**excp_type**

Controls the generation of exception data packets and/or the generation of SNMP Traps from **xmservd**. Note that these types of packets and traps can only actually be sent if **xmservd** is running. Because of this, exception packets and SNMP traps are only generated as long as

**xmservd** is active. Traps can only be generated on AIX systems. The conditions for generating exceptions and traps are controlled by the *threshold* and *frequency* parameters. The following values are valid for *excp_type*:

**SiNoHotException**
: Generate neither exceptions not traps.

**SiHotException**
: Generate exceptions but not traps.

**SiHotTrap**
: Generate SNMP traps but not exceptions.

**SiHotBoth**
: Generate both exceptions and SNMP traps.

**severity**

Required to be positive and greater than zero if exceptions are generated, otherwise specify as zero. Used to assign a severity code to the exception for display by **exmon**.

**trap_no**

Required to be positive and greater than zero if SNMP traps are generated, otherwise specify as zero. Used to assign the trap number in the generated SNMP trap.

## Return Values

The **SpmiAddSetHot** subroutine returns a pointer to a structure of type **SpmiHotVals** (“SpmiHotVals Structure” on page 210) if successful. If unsuccessful, the subroutine returns a NULL value.

## Programming Notes

The **SpmiAddSetHot** functions in a straight forward manner and as described previously in all cases where the *GrandParent* context is a context that has only one level of instantiable contexts below it. This covers most context types such as CPU, Disk, LAN, etc. In a few cases, currently only the **FS** (file system) and **RTime/ARM** (application response) contexts, the SPMI works by creating pseudo-hotvals structures that effectively expand the hotset. These pseudo-hotvals structures are created either at the time the **SpmiAddSetHot** call is issued or when new subcontexts are created for a context that’s already the *GrandParent* of a hotvals peer set. For example:

When a peer set is created for **RTime/ARM**, maybe only a few or no subcontexts of this context exists. If two applications were defined at this point, say **checking** and **savings**, one valsset would be created for the **RTime/ARM** context and a pseudo-valsset for each of **RTime/ARM/checking** and **RTime/ARM/savings**. As new applications are added to the **RTime/ARM** contexts, new pseudo-valssets are automatically added to the hotset.

Pseudo-valssets represent an implementation convenience and also helps minimize the impact of retrieving and presenting data for hotsets. As far as the caller of the **RSiGetHotItem** subroutine call is concerned, it is completely transparent. All this caller will ever see is the real hotvals structure. That is not the case for callers of **SpmiFirstHot**, **SpmiNextHot**, and **SpmiNextHotItem**. All of these subroutines will return pseudo-valssets and the calling program should be prepared to handle this.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

# SpmiCreateHotSet

## Purpose

Creates an empty hotset.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h
struct SpmiHotSet *SpmiCreateHotSet()
```

## Description

The **SpmiCreateHotSet** subroutine creates an empty hotset and returns a pointer to an **SpmiHotSet** ("SpmiHotSet Structure" on page 210) structure.This structure provides the anchor point for a hotset and must exist before the **SpmiAddSetHot** ("SpmiAddSetHot Subroutine" on page 327) subroutine can be successfully called.

## Return Values

The **SpmiCreateHotSet** subroutine returns a pointer to a structure of type **SpmiHotSet** if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- "SpmiDelSetHot Subroutine" on page 336
- "SpmiFreeHotSet Subroutine" on page 344
- "SpmiAddSetHot Subroutine" on page 327
- "Understanding SPMI Data Areas" on page 204.

# SpmiCreateStatSet Subroutine

## Purpose

Creates an empty set of statistics.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h
struct SpmiStatSet *SpmiCreateStatSet()
```

## Description

The **SpmiCreateStatSet** subroutine creates an empty set of statistics and returns a pointer to an **SpmiStatSet** ("SpmiStatSet Structure" on page 208) structure.

The **SpmiStatSet** structure provides the anchor point to a set of statistics and must exist before the **SpmiPathAddSetStat** ("SpmiPathAddSetStat Subroutine" on page 365) subroutine can be successfully called.

## Return Values

The **SpmiCreateStatSet** subroutine returns a pointer to a structure of type **SpmiStatSet** if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined

in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:


- "SpmiDelSetStat Subroutine" on page 338
- "SpmiFreeStatSet Subroutine" on page 345
- "SpmiPathAddSetStat Subroutine" on page 365
- "Understanding SPMI Data Areas" on page 204


# SpmiDdsAddCx Subroutine

## Purpose

Adds a volatile context to the contexts defined by an application.

## Library

SPMI Library (**libSpmi**.a)

## Syntax

```
#include sys/Spmidef.h

char *SpmiDdsAddCx(Ix, Path, Descr, Asnno)
ushort Ix;
char *Path, *Descr;
int Asnno;
```

## Description

The **SpmiDdsAddCx** subroutine uses the shared memory area to inform the SPMI that a context is available to be added to the context hierarchy, moves a copy of the context to shared memory, and allocates memory for the data area.

## Parameters

**Ix**

Specifies the element number of the added context in the table of dynamic contexts. No context can be added if the table of dynamic contexts has not been defined in the **SpmiDdsInit**, see "SpmiDdsInit Subroutine" on page 335 subroutine call. The first element of the table is element number 0.

**Path**

Specifies the full path name of the context to be added. If the context is not at the top-level, the parent context must already exist.

*Descr*

Provides the description of the context to be added as it will be presented to data consumers.

**Asnno**

Specifies the ASN.1 number to be assigned to the new context. All subcontexts on the same level as the new context must have unique ASN.1 numbers. Typically, each time the **SpmiDdsAddCx** subroutine adds a subcontext to the same parent context, the Asnno parameter is incremented. See “Making Dynamic Data-Supplier Statistics Unique” on page 214 for more information about ASN.1 numbers.

## Return Values

If successful, the **SpmiDdsAddCx** subroutine returns the address of the shared memory data area. If an error occurs, an error text is placed in the external **SpmiErrmsg** character array, and the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- “SpmiDdsDelCx Subroutine”
- “SpmiDdsInit Subroutine” on page 335
- “Understanding SPMI Data Areas” on page 204

---

# SpmiDdsDelCx Subroutine

## Purpose

Deletes a volatile context.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

int SpmiDdsDelCx(Area)
char *Area;
```

## Description

The **SpmiDdsDelCx** subroutine informs the SPMI that a previously added, volatile context should be deleted.

If the SPMI has not detected that the context to delete was previously added dynamically, the **SpmiDdsDelCx** subroutine removes the context from the list of to-be-added contexts and returns the allocated shared memory to the free list. Otherwise, the **SpmiDdsDelCx** subroutine indicates to the SPMI that a context and its associated statistics must be removed from the context hierarchy and any allocated shared memory must be returned to the free list.

## Parameters

**Area**

Specifies the address of the previously allocated shared memory data area as returned by an **SpmiDdsAddCx** subroutine call.

## Return Values

If successful, the **SpmiDdsDelCx** subroutine returns a value of 0. If an error occurs, an error text is placed in the external **SpmiErrmsg** character array, and the subroutine returns a nonzero value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

**/usr/include/sys/Spmidef.h** Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

## Related Information

For related information, see:

- “SpmiDdsAddCx Subroutine” on page 332
- “SpmiDdsInit Subroutine” on page 335
- “Understanding SPMI Data Areas” on page 204

## SpmiDdsInit Subroutine

## Purpose

- Establishes a program as a dynamic data-supplier (DDS) program.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

SpmiShare *SpmiDdsInit(CxTab, CxCnt, IxTab, IxCnt,
FileName)
cx_create *CxTab, *IxTab;
int CxCnt, IxCnt;
char *FileName;
```

## Description

The SpmiDdsInit subroutine establishes a program as a dynamic data-supplier (DDS) program. To do so, the SpmiDdsInit subroutine:

1. Determines the size of the shared memory required and creates a shared memory segment of that size.
2. Moves all static contexts and all statistics referenced by those contexts to the shared memory.
3. Calls the SPMI and requests it to add all of the DDS static contexts to the context tree.

**Notes:**

1. The **SpmiDdsInit** subroutine issues an **SpmiInit** subroutine call if the application program has not issued one.
2. If the calling program uses shared memory for other purposes, including memory mapping of files, the **SpmiDdsInit** or the **SpmiInit** (“SpmiInit Subroutine” on page 353) subroutine call must be issued before access is established to other shared memory areas.

## Parameters

**CxTab**

Specifies a pointer to the table of nonvolatile contexts to be added.

**CxCnt**

Specifies the number of elements in the table of nonvolatile contexts. Use the **CX_L** macro to find this value.

**IxTab**

Specifies a pointer to the table of volatile contexts the program may want to add later. If no contexts are defined, specify NULL.

**IxCnt**

Specifies the number of elements in the table of volatile contexts. Use the **CX_L** macro to find this value. If no contexts are defined, specify 0.

**FileName**

Specifies the fully qualified path and file name to use when creating the shared memory segment. At execution time, if the file exists, the process running the DDS must be able to write to the file. Otherwise, the **SpmiDdsInit** subroutine call does not succeed. If the file does not exist, it is

created. If the file cannot be created, the subroutine returns an error. If the file name includes directories that do not exist, the subroutine returns an error.

For non-AIX systems, a sixth argument is required to inform the SPMI how much memory to allocate in the DDS shared memory segment. This is not required for AIX systems because facilities exist to expand a memory allocation in shared memory. The sixth argument is:

**size**

Size in bytes of the shared memory area to allocate for the DDS program. This parameter is of type int.

## Return Values

If successful, the **SpmiDdsInit** subroutine returns the address of the shared memory control area. If an error occurs, an error text is placed in the external **SpmiErrmsg** character array, and the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:


- "SpmiExit Subroutine" on page 339
- "SpmiInit Subroutine" on page 353
- "Understanding SPMI Data Areas" on page 204


# SpmiDelSetHot Subroutine

## Purpose

Removes a single set of peer statistics from a hotset.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

int SpmiDelSetHot(HotSet, HotVal)
struct SpmiHotSet *HotSet;
struct SpmiHotVals *HotVal;
```

## Description

The **SpmiDelSetHot** subroutine removes a single set of peer statistics, identified by the *HotVal* parameter, from a hotset, identified by the *HotSet* parameter.

## Parameters

**HotSet**

Specifies a pointer to a valid structure of type **SpmiHotSet**, "SpmiHotSet Structure" on page 210, as created by the "SpmiCreateHotSet" on page 330 subroutine call.

**HotVal**

Specifies a pointer to a valid structure of type **SpmiHotVals**, see"SpmiHotVals Structure" on page 210, as created by the **SpmiAddSetHot**, see "SpmiAddSetHot Subroutine" on page 327 subroutine call. You cannot specify an **SpmiHotVals** that was internally generated by the SPMI library code as described under the *GrandParent* parameter to **SpmiAddSetHot** ("SpmiAddSetHot Subroutine" on page 327).

## Return Values

The **SpmiDelSetHot** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- "SpmiCreateHotSet" on page 330
- "SpmiFreeHotSet Subroutine" on page 344

- “SpmiAddSetHot Subroutine” on page 327
- “Understanding SPMI Data Areas” on page 204

# SpmiDelSetStat Subroutine

## Purpose

Removes a single statistic from a set of statistics.

## Library

SPMI Library (**libSpmi**.a)

## Syntax

```
#include sys/Spmidef.h

int SpmiDelSetStat(StatSet, StatVal)
struct SpmiStatSet *StatSet;
struct SpmiStatVals *StatVal;
```

## Description

The **SpmiDelSetStat** subroutine removes a single statistic, identified by the *StatVal* parameter, from a set of statistics, identified by the *StatSet* parameter.

## Parameters

**StatSet**

Specifies a pointer to a valid structure of type **SpmiStatSet**, “SpmiStatSet Structure” on page 208, as created by the **SpmiCreateStatSet** “SpmiCreateStatSet Subroutine” on page 331) subroutine call.

**StatVal**

Specifies a pointer to a valid structure of type **SpmiStatVals**, “SpmiStatVals Structure” on page 209) as created by the **SpmiPathAddSetStat**, “SpmiPathAddSetStat Subroutine” on page 365) subroutine call.

## Return Values

The **SpmiDelSetStat** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

**/usr/include/sys/Spmidef.h** Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

## Related Information

For related information, see:

- “SpmiCreateStatSet Subroutine” on page 331
- “SpmiFreeStatSet Subroutine” on page 345
- “SpmiPathAddSetStat Subroutine” on page 365
- “Understanding SPMI Data Areas” on page 204

# SpmiExit Subroutine

## Purpose

Terminates a dynamic data supplier (DDS) or local data consumer program’s association with the SPMI, and releases allocated memory.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h
void SpmiExit()
```

## Description

A successful **SpmiInit** (“SpmiInit Subroutine” on page 353) or **SpmiDdsInit** (“SpmiDdsInit Subroutine” on page 335) subroutine call allocates shared memory. Therefore, a Dynamic Data Supplier (DDS) program that has issued a successful **SpmiInit** or **SpmiDdsInit** subroutine call should issue an **SpmiExit** subroutine call before the program exits the SPMI. Allocated memory is not released until the program issues an **SpmiExit** subroutine call.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

**/usr/include/sys/Spmidef.h** Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

## Related Information

For related information, see:

- “SpmiInit Subroutine” on page 353
- “SpmiDdsInit Subroutine” on page 335

# SpmiFirstCx Subroutine

## Purpose

Locates the first subcontext of a context.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

struct SpmiCxLink *SpmiFirstCx(CxHandle)
SpmiCxHdl CxHandle;
```

## Description

The **SpmiFirstCx** subroutine locates the first subcontext of a context. The subroutine returns a NULL value if no subcontexts are found.

The structure pointed to by the returned pointer contains a handle to access the contents of the corresponding **SpmiCx** (“SpmiCx Structure” on page 206) structure through the **SpmiGetCx** (“SpmiGetCx Subroutine” on page 347) subroutine call.

## Parameters

**CxHandle**

Specifies a valid **SpmiCxHdl**, see “SpmiCxHdl Handle” on page 206 handle as obtained by another subroutine call.

## Return Values

The SpmiFirstCx subroutine returns a pointer to an **SpmiCxLink** (“SpmiCxLink Structure” on page 207) structure if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- “SpmiGetCx Subroutine” on page 347
- “SpmiNextCx Subroutine” on page 356
- “Understanding SPMI Data Areas” on page 204
- “Understanding the SPMI Data Hierarchy” on page 202

# SpmiFirstHot Subroutine

## Purpose

Locates the first of the sets of peer statistics belonging to a hotset.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

struct SpmiHotVals *SpmiFirstHot(HotSet)
struct SpmiHotSet HotSet;
```

## Description

The **SpmiFirstHot** subroutine locates the first of the **SpmiHotVals** (“SpmiHotVals Structure” on page 210) structures belonging to the specified **SpmiHotSet** (“SpmiHotSet Structure” on page 210). Using the returned pointer, the **SpmiHotSet** can then either be decoded directly by the calling program, or it can be used to specify the starting point for a subsequent **SpmiNextHotItem** (“SpmiNextHotItem Subroutine” on page 359) subroutine call. The **SpmiFirstHot** subroutine should only be executed after a successful call to the **SpmiGetHotSet** (“SpmiGetHotSet Subroutine” on page 348) subroutine.

## Parameters

**HotSet**

Specifies a valid **SpmiHotSet** structure as obtained by another subroutine call.

## Return Values

The **SpmiFirstHot** subroutine returns a pointer to a structure of type **SpmiHotVals** structure if successful. If unsuccessful, the subroutine returns a NULL value. A returned pointer may refer to a pseudo-hotvals structure as described in “Programming Notes” on page 329 for the **SpmiAddSetHot** (“SpmiAddSetHot Subroutine” on page 327) subroutine.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:


- “SpmiCreateHotSet” on page 330
- “SpmiAddSetHot Subroutine” on page 327
- “SpmiNextHot Subroutine” on page 357
- “SpmiNextHotItem Subroutine” on page 359
- “Understanding SPMI Data Areas” on page 204
- “Understanding the SPMI Data Hierarchy” on page 202


---

# SpmiFirstStat Subroutine

## Purpose

Locates the first of the statistics belonging to a context.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

struct SpmiStatLink *SpmiFirstStat(CxHandle)
SpmiCxHdl CxHandle;
```

## Description

The **SpmiFirstStat** subroutine locates the first of the statistics belonging to a context. The subroutine returns a NULL value if no statistics are found.

The structure pointed to by the returned pointer contains a handle to access the contents of the corresponding **SpmiStat** (“SpmiStat Structure” on page 206) structure through the **SpmiGetStat** (“SpmiGetStat Subroutine” on page 349) subroutine call.

## Parameters

**CxHandle**

Specifies a valid **SpmiCxHdl**, “SpmiCxHdl Handle” on page 206, handle as obtained by another subroutine call.

## Return Values

The **SpmiFirstStat** subroutine returns a pointer to a structure of type **SpmiStatLink** (“SpmiStatLink Structure” on page 207) if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:


- "SpmiGetStat Subroutine" on page 349
- "SpmiNextStat Subroutine" on page 361
- "Understanding SPMI Data Areas" on page 204
- "Understanding the SPMI Data Hierarchy" on page 202


# SpmiFirstVals Subroutine

## Purpose

Returns a pointer to the first **SpmiStatVals** structure belonging to a set of statistics.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

struct SpmiStatVals *SpmiFirstVals(StatSet)
struct SpmiStatSet *StatSet;
```

## Description

The **SpmiFirstVals** subroutine returns a pointer to the first **SpmiStatVals** ("SpmiStatVals Structure" on page 209) structure belonging to the set of statistics identified by the *StatSet* parameter. **SpmiStatVals** structures are accessed in reverse order so the last statistic added to the set of statistics is the first one returned. This subroutine call should only be issued after an **SpmiGetStatSet** ("SpmiGetStatSet Subroutine" on page 350) subroutine has been issued against the statset.

## Parameters

**StatSet**

Specifies a pointer to a valid structure of type **Spmi**StatSet as created by the **Spmi**CreateStatSet subroutine call.

## Return Values

The **SpmiFirstVals** subroutine returns a pointer to an **SpmiStatVals** structure if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:


- "SpmiCreateStatSet Subroutine" on page 331
- "SpmiNextVals Subroutine" on page 362
- "Understanding SPMI Data Areas" on page 204


---

# SpmiFreeHotSet Subroutine

## Purpose

Erases a hotset.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

int SpmiFreeHotSet(HotSet)
struct SpmiHotSet *HotSet;
```

## Description

The **SpmiFreeHotSet** subroutine erases the hotset identified by the *HotSet* parameter. All **SpmiHotVals** (“SpmiHotVals Structure” on page 210) structures chained off the **SpmiHotSet** (“SpmiHotSet Structure” on page 210) structure are deleted before the set itself is deleted.

## Parameters

**HotSet**

Specifies a pointer to a valid structure of type **SpmiHotSet** as created by the “SpmiCreateHotSet” on page 330 subroutine call.

## Return Values

The **SpmiFreeHotSet** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- “SpmiCreateHotSet” on page 330
- “SpmiDelSetHot Subroutine” on page 336
- “SpmiAddSetHot Subroutine” on page 327
- “Understanding SPMI Data Areas” on page 204

# SpmiFreeStatSet Subroutine

## Purpose

Erases a set of statistics.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

int SpmiFreeStatSet(StatSet)
struct SpmiStatSet *StatSet;
```

## Description

The **SpmiFreeStatSet** subroutine erases the set of statistics identified by the *StatSet* parameter. All **SpmiStatVals** (“SpmiStatVals Structure” on page 209) structures chained off the **SpmiStatSet** (“SpmiStatSet Structure” on page 208) structure are deleted before the set itself is deleted.

## Parameters

**StatSet**

Specifies a pointer to a valid structure of type **SpmiStatSet** as created by the **SpmiCreateStatSet**, see “SpmiCreateStatSet Subroutine” on page 331, subroutine call.

## Return Values

The **SpmiFreeStatSet** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:


- “SpmiCreateStatSet Subroutine” on page 331
- “SpmiDelSetStat Subroutine” on page 338
- “SpmiPathAddSetStat Subroutine” on page 365
- “Understanding SPMI Data Areas” on page 204

## SpmiGetCx Subroutine

### Purpose

Returns a pointer to the **SpmiCx** (“SpmiCx Structure” on page 206) structure corresponding to a specified context handle.

### Library

SPMI Library (**libSpmi.a**)

### Syntax

```
#include sys/Spmidef.h

struct SpmiCx *SpmiGetCx(CxHandle)
SpmiCxHdl CxHandle;
```

### Description

The **SpmiGetCx** subroutine returns a pointer to the **SpmiCx** structure corresponding to the context handle identified by the *CxHandle* parameter.

### Parameters

**CxHandle**

Specifies a valid **SpmiCxHdl**, see “SpmiCxHdl Handle” on page 206, handle as obtained by another subroutine call.

### Return Values

The **SpmiGetCx** subroutine returns a a pointer to an **SpmiCx** data structure if successful. If unsuccessful, the subroutine returns NULL.

### Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

### Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

### Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:


- "SpmiFirstCx Subroutine" on page 340
- "SpmiNextCx Subroutine" on page 356
- "Understanding SPMI Data Areas" on page 204
- "Understanding the SPMI Data Hierarchy" on page 202


# SpmiGetHotSet Subroutine

## Purpose

Requests the SPMI to read the data values for all sets of peer statistics belonging to a specified **SpmiHotSet** ("SpmiHotSet Structure" on page 210).

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

int SpmiGetHotSet(HotSet, Force);
struct SpmiHotSet *HotSet;
boolean Force;
```

## Description

The **SpmiGetHotSet** subroutine requests the SPMI to read the data values for all peer sets of statistics belonging to the **SpmiHotSet** identified by the *HotSet* parameter. The *Force* parameter is used to force the data values to be refreshed from their source.

The *Force* parameter works by resetting a switch held internally in the SPMI for all **SpmiStatVals** ("SpmiStatVals Structure" on page 209) and **SpmiHotVals** ("SpmiHotVals Structure" on page 210) structures, regardless of the **SpmiStatSet**s ("SpmiStatSet Structure" on page 208) and **SpmiHotSet**s to which they belong. Whenever the data value for a peer statistic is requested, this switch is checked. If the switch is set, the SPMI reads the latest data value from the original data source. If the switch is not set, the SPMI reads the data value stored in the **SpmiHotVals** structure. This mechanism allows a program to synchronize and minimize the number of times values are retrieved from the source. One method programs can use is to ensure the force request is not issued more than once per elapsed amount of time.

## Parameters

**HotSet**

Specifies a pointer to a valid structure of type **SpmiHotSet** as created by the "SpmiCreateHotSet" on page 330 subroutine call.

**Force**

If set to true, forces a refresh from the original source before the SPMI reads the data values for the set. If set to false, causes the SPMI to read the data values as they were previously retrieved from the data source.

When the force argument is set true, the effect is that of marking all statistics known by the SPMI as obsolete, which causes the SPMI to refresh all requested statistics from kernel memory or other sources. As each statistic is refreshed, the obsolete mark is reset. Statistics that are not part of the **HotSet** specified in the subroutine call remain marked as obsolete. Therefore, if an application

repetitively issues a series of, **SpmiGetHotSet** and **SpmiGetStatSet** "SpmiGetStatSet Subroutine" on page 350, subroutine calls for multiple hotsets and statsets, each time, only the first such call need set the force argument to true.

## Return Values

The **SpmiGetHotSet** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- "SpmiCreateHotSet" on page 330
- "SpmiAddSetHot Subroutine" on page 327
- "Data Access Structures and Handles, HotSets" on page 209

# SpmiGetStat Subroutine

## Purpose

Returns a pointer to the **SpmiStat** ("SpmiStat Structure" on page 206) structure corresponding to a specified statistic handle.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

struct SpmiStat *SpmiGetStat(StatHandle)
SpmiStatHdl StatHandle;
```

## Description

The **SpmiGetStat** subroutine returns a pointer to the **SpmiStat** structure corresponding to the statistic handle identified by the *StatHandle* parameter.

## Parameters

**StatHandle**

Specifies a valid **SpmiStatHdl**. see “SpmiStatHdl Handle” on page 207 handle as obtained by another subroutine call.

## Return Values

The **SpmiGetStat** subroutine returns a pointer to a structure of type **SpmiStat** if successful. If unsuccessful, the subroutine returns a NULL value.

## Return Values

The **SpmiGetStat** subroutine returns a pointer to a structure of type **SpmiStat** if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- “SpmiFirstStat Subroutine” on page 342
- “SpmiNextStat Subroutine” on page 361
- “Understanding SPMI Data Areas” on page 204
- “Understanding the SPMI Data Hierarchy” on page 202

# SpmiGetStatSet Subroutine

## Purpose

Requests the SPMI to read the data values for all statistics belonging to a specified set.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

int SpmiGetStatSet(StatSet, Force);
struct SpmiStatSet *StatSet;
boolean Force;
```

## Description

The **SpmiGetStatSet** subroutine requests the SPMI to read the data values for all statistics belonging to the **SpmiStatSet** (“SpmiStatSet Structure” on page 208) identified by the *StatSet* parameter. The *Force* parameter is used to force the data values to be refreshed from their source.

The *Force* parameter works by resetting a switch held internally in the SPMI for all **SpmiStatVals** (“SpmiStatVals Structure” on page 209) and **SpmiHotVals** (“SpmiHotVals Structure” on page 210) structures, regardless of the **SpmiStatSets** and **SpmiHotSets** (“SpmiHotSet Structure” on page 210) to which they belong. Whenever the data value for a statistic is requested, this switch is checked. If the switch is set, the SPMI reads the latest data value from the original data source. If the switch is not set, the SPMI reads the data value stored for the **SpmiStatVals** structure. This mechanism allows a program to synchronize and minimize the number of times values are retrieved from the source. One method is to ensure the force request is not issued more than once per elapsed amount of time.

## Parameters

**StatSet**

Specifies a pointer to a valid structure of type **SpmiStatSet** as created by the **SpmiCreateStatSet** (“SpmiCreateStatSet Subroutine” on page 331) subroutine call.

**Force**

If set to true, forces a refresh from the original source before the SPMI reads the data values for the set. If set to false, causes the SPMI to read the data values as they were previously retrieved from the data source.

When the force argument is set true, the effect is that of marking all statistics known by the SPMI as obsolete, which causes the SPMI to refresh all requested statistics from kernel memory or other sources. As each statistic is refreshed, the obsolete mark is reset. Statistics that are not part of the **StatSet** specified in the subroutine call remain marked as obsolete. Therefore, if an application repetitively issues the **SpmiGetStatSet** and **SpmiGetHotSet** (“SpmiGetHotSet Subroutine” on page 348) subroutine calls for multiple statsets and hotsets, each time, only the first such call need set the force argument to true.

## Return Values

The **Spmi**GetStatSet subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- "SpmiCreateStatSet Subroutine" on page 331
- "SpmiPathAddSetStat Subroutine" on page 365
- "Data Access Structures and Handles, StatSets" on page 208

---

# SpmiGetValue Subroutine

## Purpose

Returns a decoded value based on the type of data value extracted from the data field of an **SpmiStatVals** ("SpmiStatVals Structure" on page 209) structure.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

float SpmiGetValue(StatSet, StatVal)
struct SpmiStatSet *StatSet;
struct SpmiStatVals *StatVal;
```

## Description

The **SpmiGetValue** subroutine performs the following steps:

1. Verifies that an **SpmiStatVals** structure exists in the set of statistics identified by the *StatSet* parameter.
2. Determines the format of the data field as being either **SiFloat** or **SiLong** and extracts the data value for further processing.
3. Determines the data value as being of either type **SiQuantity** or type **SiCounter**.
4. If the data value is of type **SiQuantity**, returns the **val** field of the **SpmiStatVals** structure.
5. If the data value is of type **SiCounter**, returns the value of the **val_change** field of the **SpmiStatVals** structure divided by the elapsed number of seconds since the previous time a data value was requested for this set of statistics.

This subroutine call should only be issued after an **SpmiGetStatSet** ("SpmiGetStatSet Subroutine" on page 350) subroutine has been issued against the statset.

## Parameters

**StatSet**

Specifies a pointer to a valid structure of type **SpmiStatSet**, “SpmiStatSet Structure” on page 208, as created by the **SpmiCreateStatSet**, see “SpmiCreateStatSet Subroutine” on page 331, subroutine call.

**StatVal**

Specifies a pointer to a valid structure of type **SpmiStatVals** as created by the **SpmiPathAddSetStat**, “SpmiPathAddSetStat Subroutine” on page 365, subroutine call or returned by the **SpmiFirstVals** (“SpmiFirstVals Subroutine” on page 343) or **SpmiNextVals** (“SpmiNextVals Subroutine” on page 362) subroutine calls.

## Return Values

The **SpmiGetValue** subroutine returns the decoded value if successful. If unsuccessful, the subroutine returns a negative value that has a numerical value of at least 1.1.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- “SpmiGetStatSet Subroutine” on page 350
- “SpmiCreateStatSet Subroutine” on page 331
- “SpmiPathAddSetStat Subroutine” on page 365
- “Data Access Structures and Handles, StatSets” on page 208
- “Understanding SPMI Data Areas” on page 204

# SpmiInit Subroutine

## Purpose

Initializes the SPMI for a local data consumer program.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

int SpmiInit (TimeOut)
int TimeOut;
```

## Description

The **SpmiInit** subroutine initializes the SPMI. During SPMI initialization, a memory segment is allocated and the application program obtains basic addressability to that segment. An application program must issue the **SpmiInit** subroutine call before issuing any other subroutine calls to the SPMI.

**Notes:**

1. The **SpmiInit** subroutine is automatically issued by the **SpmiDdsInit** (“SpmiDdsInit Subroutine” on page 335) subroutine call. Successive **SpmiInit** subroutine calls are ignored.
2. If the calling program uses shared memory for other purposes, including memory mapping of files, the **SpmiInit** subroutine call must be issued before access is established to other shared memory areas.

The SPMI entry point called by the **SpmiInit** subroutine assigns a segment register to be used by the SPMI subroutines (and the application program) for accessing common shared memory and establishes the access mode to the common shared memory segment. After SPMI initialization, the SPMI subroutines are able to access the common shared memory segment in read-only mode.

## Parameters

**TimeOut**

Specifies the number of seconds the SPMI waits for a Dynamic Data Supplier (DDS) program to update its shared memory segment. If a DDS program does not update its shared memory segment in the time specified, the SPMI assumes that the DDS program has terminated or disconnected from shared memory and removes all contexts and statistics added by the DDS program.

The SPMI saves the largest *TimeOut* value received from the programs that invoke the SPMI. The *TimeOut* value must be zero or must be greater than or equal to 15 seconds and less than or equal to 600 seconds. A value of zero overrides any other value from any other program that invokes the SPMI and disables the checking for terminated DDS programs.

## Return Values

The **SpmiInit** subroutine returns a value of 0 if successful. If unsuccessful, the subroutine returns a nonzero value. If a nonzero value is returned, the application program should not attempt to issue additional SPMI subroutine calls.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- “SpmiDdsInit Subroutine” on page 335
- “SpmiExit Subroutine” on page 339

---

# SpmiInstantiate Subroutine

## Purpose

Explicitly instantiates the subcontexts of an instantiable context.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

int SpmiInstantiate(CxHandle)
SpmiCxHdl CxHandle;
```

## Description

The **SpmiInstantiate** subroutine explicitly instantiates the subcontexts of an instantiable context. If the context is not instantiable, do not call the **SpmiInstantiate** subroutine.

An instantiation is done implicitly by the **SpmiPathGetCx** (“SpmiPathGetCx Subroutine” on page 367) and **SpmiFirstCx** (“SpmiFirstCx Subroutine” on page 340) subroutine calls. Therefore, application programs usually do not need to instantiate explicitly.

## Parameters

**CxHandle**

Specifies a valid context handle **SpmiCxHdl**, see “SpmiCxHdl Handle” on page 206, as obtained by another subroutine call.

## Return Values

The **SpmiInstantiate** subroutine returns a value of 0 if successful. If the context is not instantiable, the subroutine returns a nonzero value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];

- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- "SpmiFirstCx Subroutine" on page 340
- "SpmiPathGetCx Subroutine" on page 367
- "Understanding the SPMI Data Hierarchy" on page 202


---

# SpmiNextCx Subroutine

## Purpose

Locates the next subcontext of a context.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h
struct SpmiCxLink *SpmiNextCx(CxLink )struct SpmiCxLink *CxLink;
```

## Description

The **SpmiNextCx** subroutine locates the next subcontext of a context, taking the context identified by the *CxLink* parameter as the current subcontext. The subroutine returns a NULL value if no further subcontexts are found.

The structure pointed to by the returned pointer contains an **SpmiCxHdl** ("SpmiCxHdl Handle" on page 206) handle to access the contents of the corresponding **SpmiCx** ("SpmiCx Structure" on page 206) structure through the **SpmiGetCx** ("SpmiGetCx Subroutine" on page 347) subroutine call.

## Parameters

### CxLink

Specifies a pointer to a valid **SpmiCxLink**, "SpmiCxLink Structure" on page 207, structure as obtained by a previous **SpmiFirstCx**, see "SpmiFirstCx Subroutine" on page 340, subroutine call.

## Return Values

The **SpmiNextCx** subroutine returns a pointer to a structure of type **SpmiCxLink** if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:


- "SpmiFirstCx Subroutine" on page 340
- "SpmiGetCx Subroutine" on page 347
- "Understanding SPMI Data Areas" on page 204.
- "Understanding the SPMI Data Hierarchy" on page 202


---

# SpmiNextHot Subroutine

## Purpose

Locates the next set of peer statistics (**SpmiHotVals** ("SpmiHotVals Structure" on page 210) structure) belonging to an **SpmiHotSet** ("SpmiHotSet Structure" on page 210).

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

struct SpmiHotVals *SpmiNextHot(HotSet, HotVals)
struct SpmiHotSet *HotSet;
struct SpmiHotVals *HotVals;
```

## Description

The **SpmiNextHot** subroutine locates the next **SpmiHotVals** structure belonging to an **SpmiHotSet**, taking the set of peer statistics identified by the *HotVals* parameter as the current one. The subroutine returns a

NULL value if no further **SpmiHotVals** structures are found. The **SpmiNextHot** subroutine should only be executed after a successful call to the **SpmiGetHotSet** (“SpmiGetHotSet Subroutine” on page 348) subroutine and (usually, but not necessarily) a call to the **SpmiFirstHot** (“SpmiFirstHot Subroutine” on page 341) subroutine and one or more subsequent calls to **SpmiNextHot**.

The subroutine allows the application programmer to position at the next set of peer statistics in preparation for using the **SpmiNextHotItem** (“SpmiNextHotItem Subroutine” on page 359) subroutine call to traverse this peer set’s array of “SpmiHotItems” on page 211 elements. Use of this subroutine is only necessary if it is desired to skip over some **SpmiHotVals** structures in an **SpmiHotSet**. Under most circumstances, the **SpmiNextHotItem** will be the sole means of accessing all elements of the “SpmiHotItems” on page 211 arrays of all peer sets belonging to an **SpmiHotSet**.

## Parameters

**HotSet**

Specifies a valid pointer to an **SpmiHotSet**, “SpmiHotSet Structure” on page 210, structure as obtained by a previous “SpmiCreateHotSet” on page 330 subroutine call.

**HotVals**

Specifies a pointer to an **SpmiHotVals**, “SpmiHotVals Structure” on page 210, structure as returned by a previous **SpmiFirstHot** or **SpmiNextHot** subroutine call or as returned by an **SpmiAddSetHot**, see “SpmiAddSetHot Subroutine” on page 327, subroutine call.

## Return Values

The **SpmiNextHot** subroutine returns a pointer to the next **SpmiHotVals** structure within the hotset. If no more **SpmiHotVals** structures are available, the subroutine returns a NULL value. A returned pointer may refer to a pseudo-hotvals structure as described in “Programming Notes” on page 329 for the **SpmiAddSetHot** (“SpmiAddSetHot Subroutine” on page 327) subroutine.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For more information, see:

- “SpmiFirstHot Subroutine” on page 341

- “SpmiGetHotSet Subroutine” on page 348
- “SpmiNextHotItem Subroutine.”
- “Data Access Structures and Handles, HotSets” on page 209

## SpmiNextHotItem Subroutine

### Purpose

Locates and decodes the next “SpmiHotItems” on page 211 element at the current position in an **SpmiHotSet** (“SpmiHotSet Structure” on page 210).

### Library

SPMI Library (**libSpmi.a**)

### Syntax

```
#include sys/Spmidef.h

struct SpmiHotVals *SpmiNextHotItem(HotSet, HotVals, index,
value, name)
struct SpmiHotSet *HotSet;
struct SpmiHotVals *HotVals;
int *index;
float *value;
char **name;
```

### Description

The **SpmiNextHotItem** subroutine locates the next **SpmiHotItems** structure belonging to an **SpmiHotSet**, taking the element identified by the *HotVals* and *index* parameters as the current one. The subroutine returns a NULL value if no further **SpmiHotItems** structures are found. The **SpmiNextHotItem** subroutine should only be executed after a successful call to the **SpmiGetHotSet** (“SpmiGetHotSet Subroutine” on page 348) subroutine.

The **SpmiNextHotItem** subroutine is designed to be used for walking all **SpmiHotItems** elements returned by a call to the **SpmiGetHotSet** subroutine, visiting the **SpmiHotVals** (“SpmiHotVals Structure” on page 210) structures one by one. By feeding the returned value and the updated integer pointed to by *index* back to the next call, this can be done in a tight loop. Successful calls to **SpmiNextHotItem** will decode each **SpmiHotItems** element and return the data value in *value* and the name of the peer context that owns the corresponding statistic in *name*.

### Parameters

**HotSet**

Specifies a valid pointer to an **SpmiHotSet**, “SpmiHotSet Structure” on page 210, structure as obtained by a previous “SpmiCreateHotSet” on page 330 subroutine call.

**HotVals**

Specifies a pointer to an **SpmiHotVals**, “SpmiHotVals Structure” on page 210, structure as returned by a previous**SpmiNextHotItem**, **SpmiFirstHot**, or **SpmiNextHot** subroutine call or as returned by an **SpmiAddSetHot** (“SpmiAddSetHot Subroutine” on page 327) subroutine call. If this parameter is specified as NULL, the first **SpmiHotVals** structure of the **SpmiHotSet** is used and the *index* parameter is assumed to be set to zero, regardless of its actual value.

**index**

A pointer to an integer that contains the desired element number in the **SpmiHotItems** array of the **SpmiHotVals** structure specified by *HotVals*. A value of zero points to the first element. When the

**SpmiNextHotItem** subroutine returns, the integer contain the index of the next **SpmiHotItems** element within the returned **SpmiHotVals** structure. If the last element of the array is decoded, the value in the integer will point beyond the end of the array, and the **SpmiHotVals** pointer returned will point to the peer set, which has now been completely decoded. By passing the returned **SpmiHotVals** pointer and the *index* parameter to the next call to **SpmiNextHotItem**, the subroutine will detect this and proceed to the first **SpmiHotItems** element of the next **SpmiHotVals** structure if one exists.

**value**

A pointer to a float variable. A successful call will return the decoded data value for the statistic. Before the value is returned, the **SpmiNextHotItem** function:

- Determines the format of the data field as being either **SiFloat** or **SiLong** and extracts the data value for further processing.
- Determines the data value as being either type **SiQuantity** or type **SiCounter** and performs one of the actions listed here:
  - If the data value is of type **SiQuantity**, the subroutine returns the **val** field of the **SpmiHotItems** structure.
  - If the data value is of type **SiCounter**, the subroutine returns the value of the **val_change** field of the **SpmiHotItems** structure divided by the elapsed number of seconds since the previous time a data value was requested for this set of statistics.

**name**

A pointer to a character pointer. A successful call will return a pointer to the name of the peer context for which the data value was read.

## Return Values

The **SpmiNextHotItem** subroutine returns a pointer to the current **SpmiHotVals** structure within the hotset. If no more **SpmiHotVals** structures are available, the subroutine returns a NULL value. The structure returned contains the data, such as threshold, which may be relevant for presentation of the results of an **SpmiGetHotSet** subroutine call to end-users. A returned pointer may refer to a pseudo-hotvals structure as described in “Programming Notes” on page 329 for the **SpmiAddSetHot** (“SpmiAddSetHot Subroutine” on page 327) subroutine.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For more information, see:

- “SpmiFirstHot Subroutine” on page 341
- “SpmiNextHot Subroutine” on page 357
- “SpmiGetHotSet Subroutine” on page 348.
- “Data Access Structures and Handles, HotSets” on page 209.

# SpmiNextStat Subroutine

## Purpose

Locates the next statistic belonging to a context.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

struct SpmiStatLink *SpmiNextStat(StatLink)
struct SpmiStatLink *StatLink;
```

## Description

The **SpmiNextStat** subroutine locates the next statistic belonging to a context, taking the statistic identified by the *StatLink* parameter as the current statistic. The subroutine returns a NULL value if no further statistics are found.

The structure pointed to by the returned pointer contains an **SpmiStatHdl** (“SpmiStatHdl Handle” on page 207) handle to access the contents of the corresponding **SpmiStat** (“SpmiStat Structure” on page 206) structure through the **SpmiGetStat** (“SpmiGetStat Subroutine” on page 349) subroutine call.

## Parameters

**StatLink**

Specifies a valid pointer to a **SpmiStatLink**, “SpmiStatLink Structure” on page 207, structure as obtained by a previous **SpmiFirstStat**, see “SpmiFirstStat Subroutine” on page 342, subroutine call.

## Return Values

The **SpmiNextStat** subroutine returns a pointer to a structure of type **SpmiStatLink** if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:

- "SpmiFirstStat Subroutine" on page 342
- "SpmiGetStat Subroutine" on page 349
- "Understanding SPMI Data Areas" on page 204.
- "Understanding the SPMI Data Hierarchy" on page 202

# SpmiNextVals Subroutine

## Purpose

Returns a pointer to the next **SpmiStatVals** ("SpmiStatVals Structure" on page 209) structure in a set of statistics.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

struct SpmiStatVals *SpmiNextVals(StatSet, StatVal)
struct SpmiStatSet *StatSet;
struct SpmiStatVals *StatVal;
```

## Description

The **SpmiNextVals** subroutine returns a pointer to the next **SpmiStatVals** structure in a set of statistics, taking the structure identified by the *StatVal* parameter as the current structure. The **SpmiStatVals** structures are accessed in reverse order so the statistic added before the current one is returned. This subroutine call should only be issued after an **SpmiGetStatSet** ("SpmiGetStatSet Subroutine" on page 350) subroutine has been issued against the statset.

## Parameters

**StatSet**

Specifies a pointer to a valid structure of type **SpmiStatSet**, "SpmiStatSet Structure" on page 208, as created by the **SpmiCreateStatSet**, see "SpmiCreateStatSet Subroutine" on page 331, subroutine call.

**StatVal**

Specifies a pointer to a valid structure of type **SpmiStatVals** as created by the **SpmiPathAddSetStat**, "SpmiPathAddSetStat Subroutine" on page 365, subroutine call or returned by a previous **SpmiFirstVals**, "SpmiFirstVals Subroutine" on page 343, or **SpmiNextVals**, "SpmiNextVals Subroutine," subroutine call.

## Return Values

The SpmiNextVals subroutine returns a pointer to a **SpmiStatVals** structure if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:


- “SpmiCreateStatSet Subroutine” on page 331
- “SpmiFirstVals Subroutine” on page 343
- “SpmiPathAddSetStat Subroutine” on page 365.
- “Data Access Structures and Handles, StatSets” on page 208


# SpmiNextValue Subroutine

## Purpose

Returns either the first **SpmiStatVals** (“SpmiStatVals Structure” on page 209) structure in a set of statistics or the next **SpmiStatVals** structure in a set of statistics and a decoded value based on the type of data value extracted from the data field of an **SpmiStatVals** structure.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

struct SpmiStatVals*SpmiNextValue( StatSet, StatVal, value)
struct SpmiStatSet *StatSet;
struct SpmiStatVals *StatVal;
float *value;
```

## Description

Instead of issuing subroutine calls to **SpmiFirstVals** (“SpmiFirstVals Subroutine” on page 343)/ **SpmiNextVals** (“SpmiNextVals Subroutine” on page 362) (to get the first or next **SpmiStatVals** structure) followed by calls to **SpmiGetValue** (“SpmiGetValue Subroutine” on page 352) (to get the decoded value from the **SpmiStatVals** structure), the **SpmiNextValue** subroutine returns both in one call. This subroutine call returns a pointer to the first **SpmiStatVals** structure belonging to the *StatSet* parameter if the *StatVal* parameter is NULL. If the *StatVal* parameter is not NULL, the next **SpmiStatVals** structure is returned, taking the structure identified by the *StatVal* parameter as the current structure. The data value corresponding to the returned **SpmiStatVals** structure is decoded and returned in the field pointed to by the value argument. In decoding the data value, the subroutine does the following:

- Determines the format of the data field as being either **SiFloat** or **SiLong** and extracts the data value for further processing.
- Determines the data value as being either type **SiQuantity** or type **SiCounter** and performs one of the actions listed here:
  - If the data value is of type **SiQuantity**, the subroutine returns the **val** field of the **SpmiStatVals** structure.
  - If the data value is of type **SiCounter**, the subroutine returns the value of the **val_change** field of the **SpmiStatVals** structure divided by the elapsed number of seconds since the previous time a data value was requested for this set of statistics.

**Note:** This subroutine call should only be issued after an **SpmiGetStatSet** (“SpmiGetStatSet Subroutine” on page 350) subroutine has been issued against the statset.

## Parameters

**StatSet**

Specifies a pointer to a valid structure of type **SpmiStatSet** as created by the **SpmiCreateStatSet**, see “SpmiCreateStatSet Subroutine” on page 331, subroutine call.

**StatVal**

Specifies either a NULL pointer or a pointer to a valid structure of type **SpmiStatVals** as created by the **SpmiPathAddSetStat**, see “SpmiPathAddSetStat Subroutine” on page 365, subroutine call or returned by a previous **SpmiNextValue** subroutine call. If *StatVal* is NULL, then the first **SpmiStatVals** pointer belonging to the set of statistics pointed to by *StatSet* is returned.

**value**A pointer used to return a decoded value based on the type of data value extracted from the data field of the returned **SpmiStatVals** structure.

## Return Value

The **SpmiNextValue** subroutine returns a pointer to a **SpmiStatVals** structure if successful. If unsuccessful, the subroutine returns a NULL value.

If the **StatVal** parameter is:

**NULL**
The first **SpmiStatVals** structure belonging to the **StatSet** parameter is returned.

**not NULL**
The next **SpmiStatVals** structure after the structure identified by the **StatVal** parameter is returned and the value parameter is used to return a decoded value based on the type of data value extracted from the data field of the returned **SpmiStatVals** structure.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Programming Notes

The **SpmiNextValue** subroutine maintains internal state information so that retrieval of the next data value from a statset can be done without traversing linked lists of data structures. The stats information is kept separate for each process, but is shared by all threads of a process.

If the subroutine is accessed from multiple threads, the state information is useless and the performance advantage is lost. The same is true if the program is simultaneously accessing two or more statsets. To benefit from the performance advantage of the **SpmiNextValue** subroutine, a program should retrieve all values in order from one stat set before retrieving values from the next statset.

The implementation of the subroutine allows a program to retrieve data values beginning at any point in the statset if the **SpmiStatVals** pointer is known. Doing so will cause a linked list traversal. If subsequent invocations of **SpmiNextValue** uses the value returned from the first and following invocation as their second argument, the traversal of the link list can be avoided.

It should be noted that the value returned by a successful **SpmiNextValue** invocation is always the pointer to the **SpmiStatVals** structure whose data value is decoded and returned in the value argument.

## Implementation Specifics

- This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:


- "SpmiGetStatSet Subroutine" on page 350
- "SpmiCreateStatSet Subroutine" on page 331
- "SpmiPathAddSetStat Subroutine."
- "Data Access Structures and Handles, StatSets" on page 208


---

# SpmiPathAddSetStat Subroutine

## Purpose

Adds a statistics value to a set of statistics.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h
struct SpmiStatVals *SpmiPathAddSetStat(StatSet, StatName,
Parent)
struct SpmiStatSet *StatSet;
char *StatName;
SpmiCxHdl Parent;
```

## Description

The **SpmiPathAddSetStat** subroutine adds a statistics value to a set of statistics. The **SpmiStatSet** ("SpmiStatSet Structure" on page 208) structure that provides the anchor point to the set must exist before the **SpmiPathAddSetStat** subroutine call can succeed.

## Parameters

**StatSet**

Specifies a pointer to a valid structure of type **SpmiStatSet** as created by the **SpmiCreateStatSet**, see "SpmiCreateStatSet Subroutine" on page 331, subroutine call.

**StatName**

Specifies the name of the statistic within the context identified by the *Parent* parameter.If the *Parent* parameter is NULL, you must specify the fully qualified path name of the statistic in the *StatName* parameter.

**Parent**

Specifies either a valid **SpmiCxHdl**, "SpmiCxHdl Handle" on page 206, handle as obtained by another subroutine call or a NULL value.

## Return Values

The **SpmiPathAddSetStat** subroutine returns a pointer to a structure of type **SpmiStatVals** ("SpmiStatVals Structure" on page 209) if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the "List of SPMI Error Codes" on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

**/usr/include/sys/Spmidef.h** Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

## Related Information

For related information, see:


- "SpmiGetStatSet Subroutine" on page 350
- "SpmiCreateStatSet Subroutine" on page 331
- "SpmiDelSetStat Subroutine" on page 338
- "SpmiFreeStatSet Subroutine" on page 345.
- "Data Access Structures and Handles, StatSets" on page 208


# SpmiPathGetCx Subroutine

## Purpose

Returns a handle to use when referencing a context.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h

SpmiCxHdl SpmiPathGetCx(CxPath, Parent)
char *CxPath;
SpmiCxHdl Parent;
```

## Description

The **SpmiPathGetCx** subroutine searches the context hierarchy for a given path name of a context and returns a handle to use when subsequently referencing the context.

## Parameters

**CxPath**

Specifies the path name of the context to find. If you specify the fully qualified path name in the *CxPath* parameter, you must set the *Parent* parameter to NULL. If the path name is not qualified or is only partly qualified (that is, if it does not include the names of all contexts higher in the data hierarchy), the **SpmiPathGetCx** subroutine begins searching the hierarchy at the context identified by the *Parent* parameter. If the *CxPath* parameter is either NULL or an empty string, the subroutine returns a handle identifying the Top context.

**Parent**

Specifies the anchor context that fully qualifies the *CxPath* parameter. If you specify a fully qualified path name in the *CxPath* parameter, you must set the *Parent* parameter to NULL.

## Return Values

The **SpmiPathGetCx** subroutine returns a handle to a context if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

**/usr/include/sys/Spmidef.h** Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI.

## Related Information

For related information, see:

- “Understanding SPMI Data Areas” on page 204
- “Understanding the SPMI Data Hierarchy” on page 202.

---

# SpmiStatGetPath Subroutine

## Purpose

Returns the full path name of a statistic.

## Library

SPMI Library (**libSpmi.a**)

## Syntax

```
#include sys/Spmidef.h>

char *miStatGetPath(Parent, StatHandle, MaxLevels)
SpmiCxHdlSp Parent;
SpmiStatHdl StatHandle;
int  MaxLevels;
```

## Description

The **SpmiStatGetPath** subroutine returns the full path name of a statistic, given a parent context **SpmiCxHdl** (“SpmiCxHdl Handle” on page 206) handle and a statistics **SpmiStatHdl** (“SpmiStatHdl Handle” on page 207) handle. The *MaxLevels* parameter can limit the number of levels in the hierarchy that must be searched to generate the path name of the statistic.

The memory area pointed to by the returned pointer is freed when the **SpmiStatGetPath** subroutine call is repeated. For each invocation of the subroutine, a new memory area is allocated and its address

returned.If the calling program needs the returned character string after issuing the **SpmiStatGetPath** subroutine call, the program must copy the returned string to locally allocated memory before reissuing the subroutine call.

## Parameters

**Parent**

Specifies a valid **SpmiCxHdl** handle as obtained by another subroutine call.

**StatHandle**

Specifies a valid **SpmiStatHdl** handle as obtained by another subroutine call. This handle must point to a statistic belonging to the context identified by the *Parent* parameter.

**MaxLevels**

Limits the number of levels in the hierarchy that must be searched to generate the path name. If this parameter is set to 0, no limit is imposed.

## Return Values

If successful, the **SpmiStatGetPath** subroutine returns a pointer to a character array containing the full path name of the statistic. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All SPMI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char SpmiErrmsg[];
- extern int SpmiErrno;

If the subroutine returns without an error, the **SpmiErrno** variable is set to 0 and the **SpmiErrmsg** character array is empty. If an error is detected, the **SpmiErrno** variable returns an error code, as defined in the **sys/Spmidef.h** file, and the **SpmiErrmsg** variable contains text, in English, explaining the cause of the error. See the “List of SPMI Error Codes” on page 242 for more information.

## Implementation Specifics

This subroutine is part of the server option of the Performance Aide for AIX licensed product and is also included in the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Spmidef.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the SPMI. |

## Related Information

For related information, see:


- “Understanding SPMI Data Areas” on page 204
- “Understanding the SPMI Data Hierarchy” on page 202.

# Appendix F. RSi Subroutines

This appendix discusses the following topics:
- "RSi Subroutines"

## RSi Subroutines

## RSiAddSetHot Subroutine

### Purpose

Add a single set of peer statistics to an already defined **SpmiHotSet** ("SpmiHotSet Structure" on page 210).

### Library

RSI Library (**libSpmi.a**)

### Syntax

```
#include sys/Rsi.h

struct SpmiHotVals *RSiAddSetHot(rhandle, HotSet, StatName,
GrandParent,
                                  maxresp, threshold, frequency, feed_type,
                                  except_type, severity, trap_no)
RSiHandle rhandle;
struct SpmiHotSet *HotSet;
char *StatName;
cx_handle GrandParent;
int maxresp;
int threshold;
int frequency;
int feed_type;
int excp_type;
int severity;
int trap_no;
```

### Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine.

**HotSet**
Specifies a pointer to a valid structure of type **SpmiHotSet** as created by the **RSiCreateHotSet** ("RSiCreateHotSet Subroutine" on page 377) subroutine call.

**StatName**
Specifies the name of the statistic within the subcontexts (peer contexts) of the context identified by the *GrandParent* parameter.

**GrandParent**
Specifies a valid **cx_handle** handle as obtained by another subroutine call. The handle must identify a context with at least one subcontext, which contains the statistic identified by the *StatName* parameter. If the context specified is one of the **RTime** contexts, no subcontext need to be created at the time the **SpmiAddSetHot** subroutine call is issued; the presence of the metric identified by the *StatName* parameter is checked against the context class description.

If the context specified has or may have multiple levels of instantiable context below it (such as the **FS** and **RTime/ARM** contexts), the metric is only searched for at the lowest context level. The **SpmiHotSet**

© Copyright IBM Corp. 1994, 2004

created is a pseudo hotvals structure used to link together a peer group of **SpmiHotVals** structures, which are created under the covers, one for each subcontext of the *GrandParent* context. In the case of **RTime/ARM**, if additional contexts are later added under the *GrandParent* contexts, additional hotsets are added to the peer group. This is transparent to the application program, except that the **RSiGetHotItem** (“RSiGetHotItem Subroutine” on page 384) subroutine call will return the peer group **SpmiHotVals** pointer rather than the pointer to the pseudo structure.

Note that specifying a specific volume group context (such as **FS/rootvg**) or a specific application context (such as **RTime/ARN/armpeek**) is still valid and won’t involve creation of pseudo **SpmiHotVals** structures.

**maxresp**
Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. If specified as zero, indicates that all “SpmiHotItems” on page 211 that meet the criteria specified by *threshold* must be returned, up-to a maximum of *maxresp* items. If both exceptions/traps and feeds are requested, the *maxresp* value is used to cap the number of exceptions/alerts as well as the number of items returned. If *feed_type* is specified as **SiHotAlways**, the *maxresp* parameter is still used to return at most *maxresp* items.

Where the *GrandParent* argument specifies a context that has multiple levels of instantiable contexts below it, the *maxresp* is applied to each of the lowest level contexts above the the actual peer contexts at a time. For example, if the *GrandParent* context is **FS** (file systems) and the system has three volume groups, then a *maxresp* value of 2 could cause up to a maximum of 2 x 3 = 6 responses to be generated.

**threshold**
Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. If specified as zero, indicates that all values read qualify to be returned in feeds. The value specified is compared to the data value read for each peer statistic. If the data value exceeds the *threshold*, it qualifies to be returned as an **SpmiHotItems** element in the **SpmiHotVals** structure. If the *threshold* is specified as a negative value, the value qualifies if it is lower than the numeric value of *threshold*. If *feed_type* is specified as **SiHotAlways**, the threshold value is ignored for feeds. For peer statistics of type **SiCounter**, the *threshold* must be specified as a rate per second; for **SiQuantity** statistics the *threshold* is specified as a level.

**frequency**
Must be non-zero if *excp_type* specifies that exceptions or SNMP traps must be generated. Ignored for feeds. Specifies the minimum number of minutes that must expire between any two exceptions/traps generated from this **SpmiHotVals** (“SpmiHotVals Structure” on page 210) structure. This value must be specified as no less than 5 minutes.

**feed_type**
Specifies if feeds of **SpmiHotItems** should be returned for this **SpmiHotVals** structure. The following values are valid:

- **SiHotNoFeed**
  No feeds should be generated
- **SiHotThreshold**
  Feeds are controlled by *threshold*.
- **SiHotAlways**
  All values, up-to a maximum of *maxresp* must be returned as feeds.

**excp_type**
Controls the generation of exception data packets and/or the generation of SNMP Traps from **xmservd**. Note that these types of packets and traps can only actually be sent if **xmservd** is running. Because of this, exception packets and SNMP traps are only generated as long as **xmservd** is active. Traps can only be generated on AIX. The conditions for generating exceptions and traps are controlled by the *threshold* and *frequency* parameters. The following values are valid for *excp_type*:

- **SiNoHotException**
  Generate neither exceptions not traps.
- **SiHotException**
  Generate exceptions but not traps.
- **SiHotTrap**
  Generate SNMP traps but not exceptions.
- **SiHotBoth**
  Generate both exceptions and SNMP traps.

**severity**
Required to be positive and greater than zero if exceptions are generated, otherwise specify as zero. Used to assign a severity code to the exception for display by **exmon**.

**trap_no**
Required to be positive and greater than zero if SNMP traps are generated, otherwise specify as zero. Used to assign the trap number in the generated SNMP trap.

## Return Values

If successful, the subroutine returns a pointer to a structure of type **struct SpmiHotVals** (“SpmiHotVals Structure” on page 210). If an error occurs, NULL is returned and an error text may be placed in the external character array **RSiEMsg**. If you attempt to add more values to a statset than the current local buffer size allows, **RSiErrno** is set to **RSiTooMany**. If you attempt to add more values than the buffer size of the remote host’s **xmservd** daemon allows, **RSiErrno** is set to **RSiBadStat** and the status field in the returned packet is set to **too_many_values**.

The external integer **RSiMaxValues** holds the maximum number of values acceptable with the data-consumer’s buffer size.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in *List of RSi Error Codes (“List of SPMI Error Codes” on page 242)* .

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:

- “RSiCreateHotSet Subroutine” on page 377
- “RSiOpen Subroutine” on page 396.

# RSiChangeFeed Subroutine

## Purpose

Changes the frequency at which the **xmservd** on the host identified by the first argument daemon is sending **data_feed** packets for a statset.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

int RSiChangeFeed(rhandle, statset, msecs)
RSiHandle rhandle;struct SpmiStatSet *statset;int msecs;
```

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

**statset**
Must be a pointer to a structure of type **struct SpmiStatSet** (“SpmiStatSet Structure” on page 208), which was previously returned by a successful **RSiCreateStatSet** (“RSiCreateStatSet Subroutine” on page 378) subroutine call. Data feeding must have been started for this **SpmiStatSet** via a previous **RSiStartFeed** (“RSiStartFeed Subroutine” on page 402) subroutine call.

**msecs**
The number of milliseconds between the sending of **data_feed** packets. This number is rounded to a multiple of **min_remote_int** milliseconds by the **xmservd** daemon on the remote host. This minimum interval can be modified through the **-i** command line interval to **xmservd**.

## Return Values

If successful, the subroutine returns zero, otherwise -1. A NULL error text is placed in the external character array **RSiEMsg** regardless of the subroutine’s success or failure.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:

- "RSiCreateStatSet Subroutine" on page 378
- "RSiOpen Subroutine" on page 396
- "RSiStartFeed Subroutine" on page 402.

# RSiChangeHotFeed Subroutine

## Purpose

Changes the frequency at which the **xmservd** on the host identified by the first argument daemon is sending **hot_feed** packets for a statset or checking if exceptions or SNMP traps should be generated.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

int RSiChangeFeed(rhandle, hotset, msecs)
RSiHandle rhandle;struct SpmiHotSet *hotset;int msecs;
```

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine.

**hotset**
Must be a pointer to a structure of type **struct SpmiHotSet** ("SpmiHotSet Structure" on page 210), which was previously returned by a successful **RsiCreateHotSet** ("RSiCreateHotSet Subroutine" on page 377) subroutine call. Data feeding must have been started for this **SpmiHotSet** via a previous **RSiStartHotFeed** ("RSiStartHotFeed Subroutine" on page 403) subroutine call.

**msecs**
The number of milliseconds between the sending of **Hot_feed** packets. This number is rounded to a multiple of **min_remote_int** milliseconds by the **xmservd** daemon on the remote host. This minimum interval can be modified through the **-i** command line interval to **xmservd**.

## Return Values

If successful, the subroutine returns zero, otherwise -1. A NULL error text is placed in the external character array **RSiEMsg** regardless of the subroutine's success or failure.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes ("List of SPMI Error Codes" on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

**/usr/include/sys/Rsi.h** Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

## Related Information

In the sample program, the **SpmiStatSet** is created in the local function **lststats** shown previously in lines 6 through 10.

- "RSiCreateHotSet Subroutine" on page 377
- "RSiOpen Subroutine" on page 396
- "RSiStartHotFeed Subroutine" on page 403.

# RSiClose Subroutine

## Purpose

Terminates the RSI interface for a remote host connection.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

void RSiClose(rhandle)
RSiHandle rhandle;
```

## Description

The **RSiClose** subroutine is responsible for:

1. Removing the data-consumer program as a known data consumer on a particular host. This is done by sending a **going_down** packet to the host.
2. Marking the RSI handle as not active.
3. Releasing all memory allocated in connection with the RSI handle.
4. Terminating the RSI interface for a remote host.

A successful **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine creates tables on the remote host it was issued against. Therefore, a data consumer program that has issued successful **RSiOpen** subroutine calls should issue an **RSiClose** ("RSiClose Subroutine") subroutine call for each RSiOpen call before the program exits so that the tables in the remote **xmservd** daemon can be released.

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** subroutine.

The macro **RSiIsOpen** can be used to test whether an RSI handle is open. It takes an **RSiHandle** as argument and returns true (1) if the handle is open, otherwise false (0).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

**/usr/include/sys/Rsi.h**
Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

## Related Information

For related information, see:

- "RSiInit Subroutine" on page 388
- "RSiOpen Subroutine" on page 396

# RSiCreateHotSet Subroutine

## Purpose

Creates an empty hotset on the remote host identified by the argument.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

struct SpmiHotSet *RSiCreateHotSet(rhandle)
RSiHandle rhandle;
```

## Description

The **RSiCreateHotSet** subroutine allocates an **SpmiHotSet** structure. The structure is initialized as an empty **SpmiHotSet** and a pointer to the **SpmiHotSet** structure is returned.

The **SpmiHotSet** structure provides the anchor point to a set of peer statistics and must exist before the **RSiAddSetHot** ("RSiAddSetHot Subroutine" on page 371) subroutine can be successfully called.

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine.

## Return Values

The **RSiCreateHotSet** subroutine returns a pointer to a structure of type **SpmiHotSet** if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes ("List of SPMI Error Codes" on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the **RSI**. |

## Related Information

For related information, see:

- "RSiOpen Subroutine" on page 396
- "RSiAddSetHot Subroutine" on page 371.

# RSiCreateStatSet Subroutine

## Purpose

Creates an empty statset on the remote host identified by the argument.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

struct SpmiStatSet *RSiCreateStatSet(rhandle)
RSiHandle rhandle;
```

## Description

The **RSiCreateStatSet** subroutine allocates an **SpmiStatSet** ("SpmiStatSet Structure" on page 208) structure. The structure is initialized as an empty **SpmiStatSet** and a pointer to the **SpmiStatSet** structure is returned.

The **SpmiStatSet** structure provides the anchor point to a set of statistics and must exist before the **RSiPathAddSetStat** ("RSiPathAddSetStat Subroutine" on page 398) subroutine can be successfully called.

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine.

## Return Values

The **RSiCreateStatSet** subroutine returns a pointer to a structure of type **SpmiStatSet** if successful. If unsuccessful, the subroutine returns a NULL value.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes ("List of SPMI Error Codes" on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the **RSI**. |

## Related Information

For related information, see:

- "RSiOpen Subroutine" on page 396
- "RSiPathAddSetStat Subroutine" on page 398.

# RSiDelSetHot Subroutine

## Purpose

Deletes a single set of peer statistics identified by an **SpmiHotVals** ("SpmiHotVals Structure" on page 210) structure from an **SpmiHotSet** ("SpmiHotSet Structure" on page 210).

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

int RSiDelSetHot(rhandle, hsp, hvp)
RSiHandle rhandle;struct SpmiHotSet *hsp;struct SpmiHotVals*hvp;
```

## Description

The **RSiDelSetHot** subroutine performs the following actions:

1. Validates that the **SpmiHotSet** identified by the second argument exists and contains the **SpmiHotVals** statistic identified by the third argument.
2. Deletes the **SpmiHotVals** value from the **SpmiHotSet** so that future **data_feed** packets do not include the deleted statistic.

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine.

**hsp**
Must be a pointer to a structure type **struct SpmiHotSet** ("SpmiHotSet Structure" on page 210), which was previously returned by a successful **RSiCreateHotSet** ("RSiCreateHotSet Subroutine" on page 377) subroutine call.

**hvp**
Must be a handle of type **struct SpmiHotVals** ("SpmiHotVals Structure" on page 210) as returned by a successful **RSiAddSetHot** ("RSiAddSetHot Subroutine" on page 371) subroutine call. You cannot specify an **SpmiHotVals** that was internally generated by the Spmi library code as described under the *GrandParent* parameter to **RSiAddSetHot** ("RSiAddSetHot Subroutine" on page 371).

## Return Values

If successful, the subroutine returns a zero value; otherwise it returns a non-zero value and an error text may be placed in the external character array **RSiEMsg**.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes ("List of SPMI Error Codes" on page 242).

### Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

### Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

### Related Information

For related information, see:

- "RSiCreateHotSet Subroutine" on page 377
- "RSiOpen Subroutine" on page 396
- "RSiAddSetHot Subroutine" on page 371.

## RSiDelSetStat Subroutine

### Purpose

Deletes a single statistic identified by an **SpmiStatVals** ("SpmiStatVals Structure" on page 209) pointer from an **SpmiStatSet** ("SpmiStatSet Structure" on page 208).

### Library

RSI Library (**libSpmi.a**)

### Syntax

```
#include sys/Rsi.h

int RSiDelSetStat(rhandle, ssp, svp)
RSiHandle rhandle;struct SpmiStatSet *ssp;struct SpmiStatVals*svp;
```

### Description

The **RSiDelSetStat** subroutine performs the following actions:

1. Validates the **SpmiStatSet** identified by the second argument exists and contains the **SpmiStatVals** statistic identified by the third argument.
2. Deletes the **SpmiStatVals** value from the **SpmiStatSet** so that future **data_feed** packets do not include the deleted statistic.

### Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine.

**ssp**
Must be a pointer to a structure type **struct SpmiStatSet** ("SpmiStatSet Structure" on page 208), which was previously returned by a successful **RSiCreateStatSet** ("RSiCreateStatSet Subroutine" on page 378) subroutine call.

**svp**
Must be a handle of type **struct SpmiStatVals** ("SpmiStatVals Structure" on page 209) as returned by a successful **RSiPathAddSetStat** ("RSiPathAddSetStat Subroutine" on page 398) subroutine call.

## Return Values

If successful, the subroutine returns a zero value; otherwise it returns a non-zero value and an error text may be placed in the external character array **RSiEMsg**.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:


- “RSiCreateStatSet Subroutine” on page 378
- “RSiOpen Subroutine” on page 396
- “RSiPathAddSetStat Subroutine” on page 398.


# RSiFirstCx Subroutine

## Purpose

Returns the first subcontext of an **SpmiCx** (“SpmiCx Structure” on page 206) context.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

struct SpmiCxLink *RSiFirstCx(rhandle, context, name,
descr)
RSiHandle rhandle;
cx_handle *context;
char **name;
char **descr;
```

## Description

The **RSiFirstCx** subroutine performs the following actions:

1. Validates that the context identified by the second argument exists.
2. Returns a handle to the first element of the list of subcontexts defined for the context.
3. Returns the short name and description of the subcontext.

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

**context**
Must be a handle of type **cx_handle**, which was previously returned by a successful **RSiPathGetCx** (“RSiPathGetCx Subroutine” on page 400) subroutine call.

**name**
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the short name of the subcontext is returned in the character array pointer.

**descr**
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the description of the subcontext is returned in the character array pointer.

## Return Values

If successful, the subroutine returns a pointer to a structure of type **struct SpmiCxLink** (“SpmiCxLink Structure” on page 207). If an error occurs or if the context doesn’t contain subcontexts, NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:

- “RSiNextCx Subroutine” on page 393
- “RSiOpen Subroutine” on page 396
- “RSiPathGetCx Subroutine” on page 400.

# RSiFirstStat Subroutine

## Purpose

Returns the first statistic of an **SpmiCx** (“SpmiCx Structure” on page 206) context.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

struct SpmiStatLink *RSiFirstStat(rhandle, context, name,
descr)
RSiHandle rhandle;
cx_handle *context;
char **name;
char **descr;
```

## Description

The **RSiFirstStat** subroutine performs the following actions:

1. Validates that the context identified by the second argument exists.
2. Returns a handle to the first element of the list of statistics defined for the context.
3. Returns the short name and description of the statistic.

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

**context**
Must be a handle of type **cx_handle**, which was previously returned by a successful **RSiPathGetCx** (“RSiPathGetCx Subroutine” on page 400) subroutine call.

**name**
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the short name of the statistics value is returned in the character array pointer.

**descr**
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the description of the statistics value is returned in the character array pointer.

## Return Values

If successful, the subroutine returns a pointer to a structure of type **struct SpmiStatLink** (“SpmiStatLink Structure” on page 207). If an error occurs, NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

**/usr/include/sys/Rsi.h** Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

## Related Information

For related information, see:


- "RSiNextStat Subroutine" on page 395
- "RSiOpen Subroutine" on page 396
- "RSiPathGetCx Subroutine" on page 400.


# RSiGetHotItem Subroutine

## Purpose

Locates and decodes the next "SpmiHotItems" on page 211 element at the current position in an incoming data packet of type **hot_feed**.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

struct SpmiHotVals *RSiGetHotItem(rhandle, HotSet, index, value,
absvalue, name)
RSiHandle rhandle;
struct SpmiHotSet **HotSet;
int *index;
float *value;
flost absvalue;
char **name;
```

## Description

The **RSiGetHotItem** subroutine locates the **SpmiHotItems** structure in the **hot_feed** data packet indexed by the value of the *index* parameter. The subroutine returns a NULL value if no further **SpmiHotItems** structures are found. The **RSiGetHotItem** subroutine should only be executed after a successful call to the **RSiGetHotSet** subroutine.

The **RSiGetHotItem** subroutine is designed to be used for walking all **SpmiHotItems** elements returned in a **hot_feed** data packet. Because the data packet may contain elements belonging to more than one **SpmiHotSet**, the *index* is purely abstract and is only used to keep position. By feeding the updated integer pointed to by *index* back to the next call, the walking of the **hot_feed** packet can be done in a tight loop. Successful calls to **RSiGetHotItem** will decode each **SpmiHotItems** element and return the data value in *value* and the name of the peer context that owns the corresponding statistic in *name*.

## Parameters

**rhandle** Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine.

**HotSet** Used to return a pointer to a valid **SpmiHotSet** ("SpmiHotSet Structure" on page 210) structure as obtained by a previous **RSiCreateHotSet** ("RSiCreateHotSet Subroutine" on page 377) subroutine call. The calling program can use this value to locate the **SpmiHotSet** if its address was stored by the program after it was created. The time stamps in the **SpmiHotSet** are updated with the time stamps of the decoded **SpmiHotItems** element.

**index** A pointer to an integer that contains the desired relative element number in the "SpmiHotItems" on page 211 array across all **SpmiStatVals** ("SpmiStatVals Structure" on page 209) contained in the data packet. A value of zero points to the first element. When the **RSiGetHotItem** subroutine returns, the integer contain the index of the next **SpmiHotItems** element in the data packet. By passing the returned *index* parameter to the next call to **RSiGetHotItem**, the calling program can iterate through all **SpmiHotItems** elements in the **hot_feed** data packet.

**value** A pointer to a float variable. A successful call will return the decoded data value of the peer statistic. Before the value is returned, the **RSiGetHotItem** function:

- Determines the format of the data field as being either **SiFloat** or **SiLong** and extracts the data value for further processing.
- Determines the data value as being either type **SiQuantity** or type **SiCounter** and performs one of the actions listed here:
  - If the data value is of type **SiQuantity**, the subroutine returns the **val** field of the **SpmiHotItems** structure.
  - If the data value is of type **SiCounter**, the subroutine returns the value of the **val_change** field of the **SpmiHotItems** structure divided by the elapsed number of seconds since the previous time a data value was requested for this set of statistics.

**absvalue** A pointer to a float variable. A successful call will return the decoded value of the **val** field of the **SpmiHotItems** structure of the peer statistic. In case of a statistic of type **SiQuantity**, this value will be the same as the one returned in the argument *value*. In case of a peer statistic of type **SiCounter**, the value returned is the absolute value of the counter.

**name** A pointer to a character pointer. A successful call will return a pointer to the name of the peer context for which the data value was read.

## Return Values

The **RSiGetHotItem** subroutine returns a pointer to the current **SpmiHotVals** ("SpmiHotVals Structure" on page 210) structure within the hotset. If no more **SpmiHotItems** elements are available, the subroutine returns a NULL value. The structure returned contains the data, such as threshold, which may be relevant for presentation of the results of an **SpmiGetHotSet** subroutine call to end-users. In the returned **SpmiHotVals** structure, all fields contain the correct values as declared, except for the following:

**stat** Declared as **SpmiStatHdl**, actually points to a valid **SpmiStat** ("SpmiStat Structure" on page 206) structure. By casting the handle to a pointer to **SpmiStat**, data in the structure can be accessed.

**grandpa** Contains the **cx_handle** for the parent context of the peer contexts.

**items** When using the **Spmi** interface this is an array of "SpmiHotItems" on page 211 structures. When using the **RSiGetHotItem** subroutine, the array is empty and attempts to access it will likely result in segmentation faults or access of not valid data.

**path** Will contain the path to the parent of the peer contexts. Even when the peer contexts are multiple levels below the parent context, the path points to the top context because the peer context identifiers in the **SpmiHotItems** elements will contain the path name from there and on. For example, if the hotvals peer set defines all volume groups, the path specified in the returned **SpmiHotVals** structure would be "**FS**" and the path name in one **SpmiHotItems** element may be "**rootvg/lv01**". When combined with the metric name from the **stat** field, the full path name can be constructed as, for example, "**FS/rootvg/lv01/%totfree**".

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes ("List of SPMI Error Codes" on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:

- “RSiOpen Subroutine” on page 396
- “RSiCreateHotSet Subroutine” on page 377.

# RSiGetRawValue Subroutine

## Purpose

Returns a pointer to a valid **SpmiStatVals** (“SpmiStatVals Structure” on page 209) structure for a given **SpmiStatVals** pointer by extraction from a **data_feed** packet. This subroutine call should only be issued from a callback function after it has been verified that a **data_feed** packet was received from the host identified by the first argument.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

struct SpmiStatVals RSiGetRawValue(rhandle, svp, index)
RSiHandle rhandle;
struct SpmiStatVals *svp;
int *index;
```

## Description

The **RSiGetRawValue** subroutine performs the following:

1. Finds an **SpmiStatVals** structure in the received data packet based upon the second argument to the subroutine call. This involves a lookup operation in tables maintained internally by the RSi interface.
2. Updates the **struct SpmiStat** pointer in the **SpmiStatVals** structure to point at a valid **SpmiStat** structure.
3. Returns a pointer to the **SpmiStatVals** structure. The returned pointer points to a static area and is only valid until the next execution of **RSiGetRawValue**.
4. Updates an integer variable with the index into the **ValsSet** array of the **data_feed** packet, which corresponds to the second argument to the call.

## Parameters

### rhandle

Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

### svp

A handle of type **struct SpmiStatVals** (“SpmiStatVals Structure” on page 209), which was previously returned by a successful **RSiPathAddSetStat** (“RSiPathAddSetStat Subroutine” on page 398) subroutine call.

**index**
A pointer to an integer variable. When the subroutine call succeeds, the index into the **ValsSet** array of the data feed packet is returned. The index corresponds to the element that matches the **svp** argument to the subroutine.

### Return Values

If successful, the subroutine returns a pointer; otherwise NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

### Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

### Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

### Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

### Related Information

For related information, see:


- “RSiGetRawValue Subroutine” on page 386
- “RSiOpen Subroutine” on page 396
- “RSiPathAddSetStat Subroutine” on page 398.


## RSiGetValue Subroutine

### Purpose

Returns a data value for a given **SpmiStatVals** (“SpmiStatVals Structure” on page 209) pointer by extraction from the **data_feed** packet. This subroutine call should only be issued from a callback function after it has been verified that a **data_feed** packet was received from the host identified by the first argument.

### Library

RSI Library (**libSpmi.a**)

### Syntax

```
#include sys/Rsi.h

float RSiGetValue(rhandle, svp)
RSiHandle rhandle;
struct SpmiStatVals *svp;
```

### Description

The **RSiGetValue** subroutine provides the following:

1. Finds an **SpmiStatVals** structure in the received data packet based upon the second argument to the subroutine call. This involves a lookup operation in tables maintained internally by the RSi interface.

2. Determines the format of the data field as being either **SiFloat** or **SiLong** and extracts the data value for further processing based upon its data format.
3. Determines the value as either of type **SiQuantity** or **SiCounter**. If the former is the case, the data value returned is the **val** field in the **SpmiStatVals** structure. If the latter type is found, the value returned by the subroutine is the **val_change** field divided by the elapsed number of seconds since the previous data packet's time stamp.

## Parameters

**rhandle**
Must be an **RSiHandle**, previously initialized by the **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine.

**svp**
A handle of type **struct SpmiStatVals** ("SpmiStatVals Structure" on page 209), which was previously returned by a successful **RSiPathAddSetStat** ("RSiPathAddSetStat Subroutine" on page 398) subroutine call.

## Return Values

If successful, the subroutine returns a non-negative value; otherwise it returns a negative value less than or equal to -1.0. A NULL error text is placed in the external character array **RSiEMsg** regardless of the subroutine's success or failure.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes ("List of SPMI Error Codes" on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:

- "RSiOpen Subroutine" on page 396
- "RSiPathAddSetStat Subroutine" on page 398

# RSiInit Subroutine

## Purpose

Allocates or changes the table of RSi handles.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

RSiHandle RSiInit(count)
int count;
```

## Description

Before any other **RSi** call is executed, a data-consumer program must issue the **RSiInit** call. Its purpose is to either:

- Allocate an array of **RSiHandleStruct** structures and return the address of the array to the data-consumer program.
- Increase the size of a previously allocated array of **RSiHandleStruct** structures and initialize the new array with the contents of the previous one.

## Parameters

**count**
Must specify the number of elements in the array of RSi handles. If the call is used to expand a previously allocated array, this argument must be larger than the current number of array elements. It must always be larger than zero. Specify the size of the array to be at least as large as the number of hosts your data-consumer program can talk to at any point in time.

## Return Values

If successful, the subroutine returns the address of the allocated array. If an error occurs, an error text is placed in the external character array **RSiEMsg** and the subroutine returns NULL. When used to increase the size of a previously allocated array, the subroutine first allocates the new array, then moves the entire old array to the new area. Application programs should, therefore, refer to elements in the RSi handle array by index rather than by address if they anticipate the need for expanding the array. The array only needs to be expanded if the number of remote hosts a data-consumer program talks to might increase over the life of the program.

An application that calls **RSiInit** repeatedly needs to preserve the previous address of the **RSiHandle** array while the **RSiInit** call is re-executed. After the call has completed successfully, the calling program should free the previous array using the **free** subroutine.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see the “RSiClose Subroutine” on page 376.

# RSiInstantiate Subroutine

## Purpose

Creates (instantiates) all subcontexts of an **SpmiCx** (“SpmiCx Structure” on page 206) context object.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

int RSiInstantiate(rhandle, context)
RSiHandle rhandle;
cx_handle *context;
```

## Description

The **RSiInstantiate** subroutine performs the following actions:

1. Validates that the context identified by the second argument exists.
2. Instantiates the context so that all subcontexts of that context are created in the context hierarchy. Note that this subroutine call currently only makes sense if the context's **SiInstFreq** is set to **SiContInst** or **SiCfgInst** because all other contexts would have been instantiated whenever the **xmservd** daemon was started.

The **RSi**Instantiate subroutine explicitly instantiates the subcontexts of an instantiable context. If the context is not instantiable, do not call the **RSi**Instantiate subroutine.

## Parameters

**rhandle**

Must point to a structure of type **RSiHandle**, which was previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

**context**

Must be a handle of type **cx_handle**, which was previously returned by a successful **RSiPathGetCx** (“RSiPathGetCx Subroutine” on page 400) subroutine call.

## Return Values

If successful, the subroutine returns a zero value; otherwise it returns an error code as defined in **SiError** and an error text may be placed in the external character array **RSiEMsg**.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:

- “RSiFirstCx Subroutine” on page 381
- “RSiOpen Subroutine” on page 396
- “RSiPathGetCx Subroutine” on page 400.

# RSiInvite Subroutine

## Purpose

Invites data suppliers on the network to identify themselves and returns a table of data-supplier host names.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

char **RSiInvite(resy_callb, excp_callb)
int (*resy_callb)();
int (*excp_callb)();
```

## Description

The **RSiInvite** subroutine call broadcasts **are_you_there** messages on the network to provoke **xmservd** daemons on remote hosts to respond and returns a table of all responding hosts.

## Parameters

The arguments to the subroutine are:

**resy_callb**
Must be either NULL or a pointer to a function that processes **i_am_back** packets as they are received from the **xmservd** daemons on remote hosts for the duration of the **RSiInvite** subroutine call. When the callback function is invoked, it is passed three arguments as described in the following information.

If this argument is specified as NULL, a callback function internal to the **RSiInvite** subroutine receives any **i_am_back** packets and uses them to build the table of host names the function returns.

**excp_callb**
Must be NULL or a pointer to a function that processes **except_rec** packets as they are received from the **xmservd** daemons on remote hosts. If a NULL pointer is passed, your application does not receive **except_rec** messages. When this callback function is invoked, it is passed three arguments as described in the following information.

This argument always overrides the corresponding argument of any previous **RSiInvite** or **RSiOpen** call, and it can be overridden by subsequent executions of either. In this way, your application can turn exception monitoring on and off. For an **RSiOpen** to override the exception processing specified by a previous open call, the connection must first be closed with the **RSiClose** call. That’s because an **RSiOpen** against an already active handle is treated as a no-operation.

The **resy_callb** and **excp_callb** functions in your application are called with the following three arguments:

- An **RSiHandle**. The RSi handle pointed to is almost certain not to represent the host that sent the packet. Ignore this argument, and use only the second one: the pointer to the input buffer.
- A pointer of type **pack *** to the input buffer containing the received packet. Always use this pointer rather than the pointer in the **RSiHandle** structure.

- A pointer of type **struct sockaddr_in *** to the IP address of the originating host.

## Return Values

If successful, the subroutine returns an array of character pointers, each of which contains a host name of a host that responded to the invitation. The returned host names are actually constructed as two "words" with the first one being the host name returned by the host in response to an **are_you_there** request; the second one being the character form of the host's IP address. The two "words" are separated by one or more blanks. This format is suitable as an argument to the **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine call. In addition, the external integer variable **RSiInvTabActive** contains the number of host names found. The returned pointer to an array of host names must not be freed by the subroutine call. The calling program should not assume that the pointer returned by this subroutine call remains valid after subsequent calls to **RSiInvite**. If the call is not successful, an error text is placed in the external character array **RSiEMsg**, an error number is placed in **RSiErrno**, and the subroutine returns NULL.

The list of host names returned by **RSiInvite** does not include the hosts your program has already established a connection with through an **RSiOpen** call. Your program is responsible for keeping track of such hosts. If you need a list of both sets of hosts, either let the **RSiInvite** call be the first one issued from your program or merge the list of host names returned by the call with the list of hosts to which you have connections.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes ("List of SPMI Error Codes" on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see"RSiOpen Subroutine" on page 396.

# RSiMainLoop Subroutine

## Purpose

Allows an application to suspend execution and wait to get awakened when data feeds arrive.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

void RSiMainLoop(msecs)
int msecs;
```

## Description

The **RSiMainLoop** subroutine:

1. Allows the data-consumer program to suspend processing while waiting for **data_feed** packets to arrive from one or more **xmservd** daemons.
2. Tells the subroutine that waits for data feeds to return control to the data-consumer program so that the latter can check for and react to other events.
3. Invokes the subroutine to process **data_feed** packets for each such packet received.

To work properly, the **RSiMainLoop** subroutine requires that at least one **RSiOpen** (“RSiOpen Subroutine” on page 396) call has been successfully completed and that the connection has not been closed.

## Parameters

**msecs**

The minimum elapsed time in milliseconds that the subroutine should continue to attempt receives before returning to the caller. Notice that your program releases control for as many milliseconds you specify but that the callback functions defined on the **RSiOpen** call may be called repetitively during that time.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see “RSiOpen Subroutine” on page 396.

# RSiNextCx Subroutine

## Purpose

Returns the next subcontext of an **SpmiCx** (“SpmiCx Structure” on page 206) context.

## Library

RSI Library (**libSpmi**.a)

## Syntax

```
#include sys/Rsi.h

struct SpmiCxLink *RSiNextCx(rhandle, context, link, name,
descr)
RSiHandle rhandle;
```

```
cx_handle *context;
struct SpmiCxLink *link;
char **name;
char **descr;
```

## Description

The **RSiNextCx** subroutine:

1. Validates that the context identified by the second argument exists.
2. Returns a handle to the next element of the list of subcontexts defined for the context.
3. Returns the short name and description of the subcontext.

## Parameters

**rhandle**
Must point to a structure of type **RSiHandle**, which was previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

**context**
Must be a handle of type **cx_handle**, which was previously returned by a successful **RSiPathGetCx** (“RSiPathGetCx Subroutine” on page 400) subroutine call.

**link**
Must be a pointer to a structure of type **struct SpmiCxLink** (“SpmiCxLink Structure” on page 207), which was previously returned by a successful **RSiFirstCx** (“RSiFirstCx Subroutine” on page 381) or **RSiNextCx** (“RSiNextCx Subroutine” on page 393) subroutine call.

**name**
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the short name of the subcontext is returned in the character array pointer.

**descr**
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the description of the subcontext is returned in the character array pointer.

## Return Values

If successful, the subroutine returns a pointer to a structure of type **struct SpmiCxLink** (“SpmiCxLink Structure” on page 207). If an error occurs, or if no more subcontexts exist for the context, NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

**/usr/include/sys/Rsi.h** Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI.

## Related Information

For related information, see:

- "RSiFirstCx Subroutine" on page 381
- "RSiOpen Subroutine" on page 396
- "RSiPathGetCx Subroutine" on page 400.

# RSiNextStat Subroutine

## Purpose

Returns the next statistic of an **SpmiCx** ("SpmiCx Structure" on page 206) context.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h
```

```
struct SpmiStatLink *RSiNextStat(rhandle, context, link, name,
descr)
RSiHandle rhandle;
cx_handle *context;
struct SpmiStatLink *link;
char **name;
char **descr;
```

## Description

The **RSiNextStat** subroutine:

1. Validates that a context identified by the second argument exists.
2. Returns a handle to the next element of the list of statistics defined for the context.
3. Returns the short name and description of the statistic.

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine.

**context**
Must be a handle of type **cx_handle**, which was previously returned by a successful **RSiPathGetCx** ("RSiPathGetCx Subroutine" on page 400) subroutine call.

**link**
Must be a pointer to a structure of type **struct SpmiStatLink** ("SpmiStatLink Structure" on page 207), which was previously returned by a successful **RSiFirstStat** ("RSiFirstStat Subroutine" on page 382) or **RSiNextStat** subroutine call.

**name**
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the short name of the statistics value is returned in the character array pointer.

**descr**
Must be a pointer to a pointer to a character array. The pointer must be initialized to point at a character array pointer. When the subroutine call is successful, the description of the statistics value is returned in the character array pointer.

## Return Values

If successful, the subroutine returns a pointer to a structure of type **struct SpmiStatLink**. If an error occurs, or if no more statistics exists for the context, NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:


- “RSiFirstStat Subroutine” on page 382
- “RSiOpen Subroutine”
- “RSiPathGetCx Subroutine” on page 400.


# RSiOpen Subroutine

## Purpose

Initializes the RSi interface for a remote host.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

int RSiOpen(rhandle, wait, bufsize, hostID, feed_callb,
            resy_callb, excp_callb)
RSiHandle rhandle;
int wait;
int bufsize;
char *hostID;
int (*feed_callb)();
int (*resy_callb)();
int (*excp_callb)();
```

## Description

The **RSiOpen** subroutine performs the following actions:

1. Establishes the issuing data-consumer program as a data consumer known to the **xmservd** daemon on a particular host. The subroutine does this by sending an **are_you_there** packet to the host.
2. Initializes an RSi handle for subsequent use by the data-consumer program.

## Parameters

The arguments to the subroutine are:

**rhandle**
Must point to an element of the **RSiHandleStruct** array, which is returned by a previous **RSiInit** ("RSiInit Subroutine" on page 388) call. If the subroutine is successful the structure is initialized and ready to use as a handle for subsequent RSi interface subroutine calls.

**wait**
Must specify the timeout in milliseconds that the RSi interface shall wait for a response when using the request-response functions. On LANs, a reasonable value for this argument is 100 milliseconds. If the response is not received after the specified wait time, the library subroutines retry the receive operation until five times the wait time has elapsed before returning a timeout indication. The wait time must be zero or more milliseconds.

**bufsize**
Specifies the maximum buffer size to be used for constructing network packets. This size must be at least 4,096 bytes. The buffer size determines the maximum packet length that can be received by your program and sets the limit for the number of data values that can be received in one **data_feed** packet. There's no point in setting the buffer size larger than that of the **xmservd** daemon because both must be able to handle the packets. If you need large sets of values, you can use the command line argument **-b** of **xmservd** to increase its buffer size up to 16,384 bytes.

The fixed part of a **data_feed** packet is 104 bytes and each value takes 32 bytes. A buffer size of 4,096 bytes allows up to 124 values per packet.

**hostID**
Must be a character array containing the identification of the remote host whose **xmservd** daemon is the one with which you want to talk. The first characters of the host identification (up to the first white space) is used as the host name. The full host identification is stored in the **RSiHandle** field **longname** and may contain any description that helps the end user identify the host used. The host name may be either in long format (including domain name) or in short format.

**feed_callb**
Must be a pointer to a function that processes **data_feed** packets as they are received from the **xmservd** daemon. When this callback function is invoked, it is passed three arguments as described in the following information.

**resy_callb**
Must be a pointer to a function that processes **i_am_back** packets as they are received from the **xmservd** daemon. When this callback function is invoked it is passed three arguments as described in the following information.

**excp_callb**
Must be NULL or a pointer to a function that processes **except_rec** packets as they are received from the **xmservd** daemon. If a NULL pointer is passed, your application does not receive **except_rec** messages. When this callback function is invoked, it is passed three arguments as described in the following information. This argument always overrides the corresponding argument of any previous **RSiInvite** ("RSiInvite Subroutine" on page 391) or **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine call and

can itself be overridden by subsequent executions of either. In this way, your application can turn exception monitoring on and off. For an **RSiOpen** call to override the exception processing specified by a previous open call, the connection must first be closed with the **RSiClose** (“RSiClose Subroutine” on page 376) subroutine call.

The **feed_callb**, **resy_callb**, and **excp_callb** functions are called with the arguments:

**RSiHandle**. When a **data_feed** packet is received, the structure pointed to is guaranteed to represent the host sending the packet. In all other situations the **RSiHandle** structure may represent any of the hosts to which your application is talking.

Pointer of type **pack *** to the input buffer containing the received packet. In callback functions, always use this pointer rather than the pointer in the **RSiHandle** structure.

Pointer of type **struct sockaddr_in *** to the IP address of the originating host.

### Return Values

If successful, the subroutine returns zero and initializes the array element of type **RSiHandle** pointed to by **rhandle**. If an error occurs, error text is placed in the external character array **RSiEMsg** and the subroutine returns a negative value.

### Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

### Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

### Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

### Related Information

For related information, see:

- “RSiClose Subroutine” on page 376
- “RSiInvite Subroutine” on page 391
- “RSiOpen Subroutine” on page 396.

## RSiPathAddSetStat Subroutine

### Purpose

Add a single statistics value to an already defined **SpmiStatSet (“SpmiStatSet Structure” on page 208)**.

### Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

struct SpmiStatVals *RSiPathAddSetStat(rhandle, statset,
path)
RSiHandle rhandle;
struct SpmiStatSet *statset;
char *path;
```

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

**statset**
Must be a pointer to a structure of type **struct SpmiStatSet**, which was previously returned by a successful **RSiCreateStatSet** (“RSiCreateStatSet Subroutine” on page 378) subroutine call.

**path**
Must be the full value path name of the statistics value to add to the **SpmiStatSet**. The value path name must not include a terminating slash. Note that value path names never start with a slash.

## Return Values

If successful, the subroutine returns a pointer to a structure of type **struct SpmiStatVals** (“SpmiStatVals Structure” on page 209). If an error occurs, NULL is returned and an error text may be placed in the external character array **RSiEMsg**. If you attempt to add more values to a statset than the current local buffer size allows, **RSiErrno** is set to **RSiTooMany**. If you attempt to add more values than the buffer size of the remote host’s **xmservd** daemon allows, **RSiErrno** is set to **RSiBadStat** and the status field in the returned packet is set to **too_many_values**.

The external integer **RSiMaxValues** holds the maximum number of values acceptable with the data-consumer’s buffer size.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:

- “RSiCreateStatSet Subroutine” on page 378
- “RSiOpen Subroutine” on page 396.

# RSiPathGetCx Subroutine

## Purpose

Searches the context hierarchy for an **SpmiCx** (“SpmiCx Structure” on page 206) context that matches a context path name.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

cx_handle *RSiPathGetCx(rhandle, path)
RSiHandle rhandle;
char *path;
```

## Description

The **RSiPathGetCx** subroutine performs the following actions:

1. Searches the context hierarchy for a given path name of a context.
2. Returns a handle to be used when subsequently referencing the context.

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

**path**
A path name of a context for which a handle is to be returned. The context path name must be the full path name and must not include a terminating slash. Note that context path names never start with a slash.

## Return Values

If successful, the subroutine returns a handle defined as a pointer to a structure of type **cx_handle**. If an error occurs, NULL is returned and an error text may be placed in the external character array **RSiEMsg**.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:

- “RSiFirstCx Subroutine” on page 381
- “RSiOpen Subroutine” on page 396
- “RSiNextCx Subroutine” on page 393.

# RSiStatGetPath Subroutine

## Purpose

Finds the full path name of a statistic identified by a **SpmiStatVals** (“SpmiStatVals Structure” on page 209) pointer.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

char *RSiStatGetPath(rhandle, svp)
RSiHandle rhandle;
struct SpmiStatVals *svp;
```

## Description

The **RSiStatGetPath** subroutine performs the following:

1. Validates that the **SpmiStatVals** statistic identified by the second argument does exist.
2. Returns a pointer to a character array containing the full value path name of the statistic.

The memory area pointed to by the returned pointer is freed when the **RSiStatGetPath** subroutine call is repeated. For each invocation of the subroutine, a new memory area is allocated and its address returned.

If the calling program needs the returned character string after issuing the **RSiStatGetPath** subroutine call, the program must copy the returned string to locally allocated memory before reissuing the subroutine call.

## Parameters

**rhandle**
Must be an **RSiHandle**, previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

**svp**
Must be a handle of type **struct SpmiStatVals** as returned by a successful **RSiPathAddSetStat** (“RSiPathAddSetStat Subroutine” on page 398) subroutine call.

## Return Values

If successful, the **RSiStatGetPath** subroutine returns a pointer to a character array containing the full path name of the statistic. If unsuccessful, the subroutine returns a NULL value and an error text may be placed in the external character array **RSiEMsg**.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

### Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

### Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

### Related Information

For related information, see:

- “RSiOpen Subroutine” on page 396
- “RSiPathAddSetStat Subroutine” on page 398.

## RSiStartFeed Subroutine

### Purpose

Tells **xmservd** to start sending data feeds for a statset.

### Library

RSI Library (**libSpmi.a**)

### Syntax

```
#include sys/Rsi.h

int RSiStartFeed(rhandle, statset, msecs)
RSiHandle rhandle;
struct SpmiStatSet *statset;
int msecs;
```

### Description

The **RSiStartFeed** subroutine performs the following function:

1. Informs **xmservd** of the frequency with which it is required to send **data_feed** packets.
2. Tells the **xmservd** to start sending **data_feed** packets.

### Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

**statset**
Must be a pointer to a structure of type **struct SpmiStatSet** (“SpmiStatSet Structure” on page 208), which was previously returned by a successful **RSiCreateStatSet** (“RSiCreateStatSet Subroutine” on page 378) subroutine call.

**msecs**
The number of milliseconds between the sending of **data_feed** packets. This number is rounded to a multiple of **min_remote_int** milliseconds by the **xmservd** daemon on the remote host. This minimum interval can be modified through the **-i** command line interval to **xmservd**.

## Return Values

If successful, the subroutine returns zero; otherwise it returns -1 and an error text may be placed in the external character array **RSiEMsg**.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes ("List of SPMI Error Codes" on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:


- "RSiCreateStatSet Subroutine" on page 378
- "RSiOpen Subroutine" on page 396
- "RSiStopFeed Subroutine" on page 404.


# RSiStartHotFeed Subroutine

## Purpose

Tells **xmservd** to start sending hot feeds for a hotset or to start checking for if exceptions or SNMP traps should be generated.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

int RSiStartFeed(rhandle, hotset, msecs)
RSiHandle rhandle;
struct SpmiHotSet *hotset;
int msecs;
```

## Description

The **RSiStartHotFeed** subroutine performs the following function:

1. Informs **xmservd** of the frequency with which it is required to send **hot_feed** packets, if the hotset is defined to generate **hot_feed** packets.
2. Informs **xmservd** of the frequency with which it is required to check if exceptions or SNMP traps should be generated. This is only done if it is specified for the hotset that exceptions and/or SNMP traps should be generated.
3. Tells the **xmservd** to start sending **data_feed** packets and/or start checking for exceptions or traps.

## Parameters

**rhandle**
Must be an **RSiHandle**, which was previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

**hotset**
Must be a pointer to a structure of type **struc SpmiHotSet** (“SpmiHotSet Structure” on page 210), which was previously returned by a successful **RSiCreateHot** (“RSiCreateHotSet Subroutine” on page 377) subroutine call.

**msecs**
The number of milliseconds between the sending of **hot_feed** packets and/or the number of milliseconds between checks for if exceptions or SNMP traps should be generated. This number is rounded to a multiple of **min_remote_int** milliseconds by the **xmservd** daemon on the remote host. This minimum interval can be modified through the **-i** command line interval to **xmservd**.

## Return Values

If successful, the subroutine returns zero; otherwise it returns -1 and an error text may be placed in the external character array **RSiEMsg**.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSiErrno** variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:

- “RSiCreateHotSet Subroutine” on page 377
- “RSiOpen Subroutine” on page 396
- “RSiChangeHotFeed Subroutine” on page 375
- “RSiStopHotFeed Subroutine” on page 406.

# RSiStopFeed Subroutine

## Purpose

Tells **xmservd** to stop sending data feeds for a statset.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

int RSiStopFeed(rhandle, statset, erase)
RSiHandle rhandle;
struct SpmiStatSet *statset;
boolean erase;
```

## Description

The **RSiStopFeed** subroutine instructs the **xmservd** of a remote system to:

1. Stop sending **data_feed** packets for a given **SpmiStatSet** (“SpmiStatSet Structure” on page 208). If the daemon is not told to erase the **SpmiStatSet**, feeding of data can be resumed by issuing the **RSiStartFeed** (“RSiStartFeed Subroutine” on page 402) subroutine call for the **SpmiStatSet**.
2. Optionally tells the daemon and the API library subroutines to erase all their information about the **SpmiStatSet**. Subsequent references to the erased **SpmiStatSet** are not valid.

## Parameters

**rhandle**
Must point to a structure of type **RSiHandle**, which was previously initialized by the **RSiOpen** (“RSiOpen Subroutine” on page 396) subroutine.

**statset**
Must be a pointer to a structure of type **struct SpmiStatSet**, which was previously returned by a successful **RSiCreateStatSet** (“RSiCreateStatSet Subroutine” on page 378) subroutine call. Data feeding must have been started for this **SpmiStatSet** via a previous **RSiStartFeed** (“RSiStartFeed Subroutine” on page 402) subroutine call.

**erase**
If this argument is set to true, the **xmservd** daemon on the remote host discards all information about the named **SpmiStatSet**. Otherwise the daemon maintains its definition of the set of statistics.

## Return Values

If successful, the subroutine returns zero, otherwise -1. A NULL error text is placed in the external character array **RSiEMsg** regardless of the subroutine’s success or failure.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSi**Errno variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes (“List of SPMI Error Codes” on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:

- "RSiOpen Subroutine" on page 396
- "RSiStartFeed Subroutine" on page 402.

# RSiStopHotFeed Subroutine

## Purpose

Tells **xmservd** to stop sending hot feeds for a hotset and to stop checking for exception and SNMP trap generation.

## Library

RSI Library (**libSpmi.a**)

## Syntax

```
#include sys/Rsi.h

int RSiStopFeed(rhandle, hotset, erase)
RSiHandle rhandle;
struct SpmiHotSet *hotset;
boolean erase;
```

## Description

The **RSiStopHotFeed** subroutine instructs the **xmservd** of a remote system to:

1. Stop sending **hot_feed** packets or check if exceptions or SNMP traps should be generated for a given **SpmiHotSet** ("SpmiHotSet Structure" on page 210). If the daemon is not told to erase the **SpmiHotSet**, feeding of data can be resumed by issuing the **RSiStartHotFeed** ("RSiStartHotFeed Subroutine" on page 403) subroutine call for the **SpmiHotSet**.
2. Optionally tells the daemon and the API library subroutines to erase all their information about the **SpmiHotSet**. Subsequent references to the erased **SpmiHotSet** are not valid.

## Parameters

**rhandle**
Must point to a structure of type **RSiHandle**, which was previously initialized by the **RSiOpen** ("RSiOpen Subroutine" on page 396) subroutine.

**hotset**
Must be a pointer to a structure of type **struct SpmiHotSet**, which was previously returned by a successful **RSiCreateHotSet** ("RSiCreateHotSet Subroutine" on page 377) subroutine call. Data feeding must have been started for this **SpmiStatSet** via a previous **RSiStartHotFeed** ("RSiStartHotFeed Subroutine" on page 403) subroutine call.

**erase**
If this argument is set to true, the **xmservd** daemon on the remote host discards all information about the named **SpmiHotSet**. Otherwise the daemon maintains its definition of the set of statistics.

## Return Values

If successful, the subroutine returns zero, otherwise -1. A NULL error text is placed in the external character array **RSiEMsg** regardless of the subroutine's success or failure.

## Error Codes

All RSI subroutines use external variables to provide error information. To access these variables, an application program must define the following external variables:

- extern char RSiEMsg[];
- extern int RSiErrno;

If the subroutine returns without an error, the **RSiErrno** variable is set to **RSiOkay** and the **RSiEMsg** character array is empty. If an error is detected, the **RSi**Errno variable returns an error code, as defined in the enum **RSiErrorType**. RSi error codes are described in List of RSi Error Codes ("List of SPMI Error Codes" on page 242).

## Implementation Specifics

This subroutine is part of the Performance Toolbox for AIX licensed product.

## Files

| | |
|---|---|
| **/usr/include/sys/Rsi.h** | Declares the subroutines, data structures, handles, and macros that an application program can use to access the RSI. |

## Related Information

For related information, see:


- "RSiOpen Subroutine" on page 396
- "RSiStartHotFeed Subroutine" on page 403
- "RSiChangeHotFeed Subroutine" on page 375.

# Appendix G. Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Dept. LRAS/Bldg. 003
11400 Burnet Road
Austin, TX 78758-3498
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

© Copyright IBM Corp. 1994, 2004

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrates programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:
(c) (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. (c) Copyright IBM Corp. _enter the year or years_. All rights reserved.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

## Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, other countries, or both:

AIX
CUA
IBM
RS/6000
pSeries

NetView

Java and all Java-based trademarks and logos are registered trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be the trademarks or service marks of others.

# Glossary

**agent.** Data collecting component of PTX.

**azizo.** A powerful tool used to analyze performance recordings.

**cascading menu.** A submenu of related choices that is invoked when the parent item, is selected. Usually, a choice that offers a cascading menu is designated by an arrow to the right of the choice. *Similar to a context line.*

**context line.** Menu items ending in a slash and three dots (/...). The slash and three dots signify that the line itself represents a list at the next hierarchical level. *Contrast with statistics lines.*

**console.** A customizable graphical window monitoring component of xmperf.

**data-consumer.** A description of a host, program, or such that receives statistics over the network from the **xmservd** daemon and prints, post-processes, or otherwise manipulates the raw statistics. Synonymous with client. *Contrast with data-supplier.*

**data-supplier host.** Describes a host, program, or such that supplies statistics across a network. Synonymous with server. *Contrast with data-consumer.*

**ghost instrument.** An empty space in the console where an instrument used to be. Usually caused when a console designed for one system contains instruments not available on the current system. Ghost instruments occupy the space and prevent you from defining a new instrument in that same space and moving or resizing other instruments to use the space.

**ghosted.** A description of an unavailable choice. Menu items are *ghosted* to indicate that a standard choice is not available under the current circumstances.

**instrument.** The actual monitoring device enclosed within consoles as rectangular graphical subwindows.

**localhost.** The assumed host when no hostname is given.

**manager.** The analytical, monitoring, and display component of PTX.

**metric.**A probe in or instrumentation of a component of the operating system.

**pixel.** An abbreviation for picture element.

**pixel map.** A three-dimensional array of bits. A pixel map can be thought of as a two-dimensional array of pixels, with each pixel being a value from zero to 2 to the power N -1, where N is the depth of the pixel map.

**pixmap.** (1) A data type to which icons, originally created as bitmaps, are converted. After this conversion, the appropriate subroutines can generate pixmaps through references to a defaults file, by name, and through an argument list, by pixmap. (2) An abbreviation for pixel map. *See pixel map.*

**radio button.** Indicates a fixed set of choices. Only one of the buttons in the set can be selected at a time. A circle with text beside it. The circle is partially filled when a choice is selected.

**Remote Statistics Interface (RSi).** The Manager API which allows an application program to access statistics from remote nodes (or the local host) through a network interface.

**rmss.** A tool used to simulate different real memory sizes.

**statistic line.** The lines in a list that represent a specific value. *Contrast with context line.*

**SiCounter**. A value that's incremented continuously. Instruments show the delta (change) in the value between observations, divided by the elapsed time, representing a rate per second.

**SiQuantity value**. Represents a level, such as memory used or available disk space. The actual observation value is shown by instruments.

**statistic.**A probe in or instrumentation of a component of the operating system.

**stop record.** A special type of value record which signals that recording was stopped for a statset and gives the time it happened. This allows

© Copyright IBM Corp. 1994, 2004

programs using the recording file to distinguish between gaps in the recording and variances in recording interval.

**System Performance Measurement Interface (Spmi).** The Agent API which allows an application program to register custom performance statistics about its own performance or that of some other system component. Once registered, the custom statistics become available to any consumer of statistics, local or remote. Also permits applications to access statistics on the local system without using the network interface. Such applications are called local data-consumer programs.

**tabulating windows.** Special forms of windows that tabulate the values of an instrument as data is received and will also calculate a line with a weighted average for each value.

**tprof.** A tool used to determine which part of a program most of the execution time is spent.

**value.** Used to refer to a statistic (metric) when included in a monitoring device.

**xmpeek.** A program that allows you to ask any host about the status of its **xmservd** daemon.

# Index

## Numerics


3dmon 71, 290
  autoscaling 73
  command line 74
  configuration file 76
  customizing 76
  dual wildcard configuration 78
  exiting 76
  hardware dependencies 76
  how to record 73
  menus 72
  Monitoring ARM metrics from 199
  monitoring IP response time 194
  overview 71
  path name display 74
  recording 80
  resynchronizing with multiple hosts 73
  Rsi.hosts file 79
  single wildcard configuration 76
  user interface 71
  viewing obscured statistics 73
  X Resources 79
3dplay 83, 292
  command line invocation 84
  invocation from 3dmon 84
  invocation from xmperf 84
  overview 83
  user interface 84


## A


a2ptx
  command line 96
  data values 96
  formatting files 95
  host identifier 95
  input file format 95
  recording generator 95
  statistic names 96
  time stamps 96
add instrument submenu
  xmperf 43
agent 3
agent installation 268
alarm definition
  filtd 187
alarms with filtd 183
All skeleton type 16
analysis and control 2
annotations
  xmperf 58
    types and fields 58
    using 59
    while recording 59
application programming interfaces 3
application response time measurement 195
  ARM contexts in Spmi data space 195
application response time measurement *(continued)*
  ARM implementation restrictions 196
  ARM library implementation 196
  ARM run-time control 197
  ARM transaction metrics 196
  SpmiArmd daemon 197
    configuring 198
applying configurations 124
ARM Subroutines 305
  arm_end 313
  arm_end Dual Call 323
  arm_getid 306
  arm_getid Dual Call 316
  arm_init 305
  arm_init Dual Call 314
  arm_start 308
  arm_start Dual Call 319
  arm_stop 311
  arm_stop Dual Call 322
  arm_update 310
  arm_update Dual Call 320
azizo
  adding metrics to main graphs 120
  annotate icon 130
  changing the appearance of main graphs 121
  changing the style of metrics 117
  command line 110
  config icon 130
  configuration file 123
  configuration lines 130
  dialog boxes 125
  drag and drop operations 130
  exit icon 131
  exiting 113
  filter icon 131
  filtered recordings 122
  filtering files 94
  help facility 113
  help icon 131
  icon section 108
  info icon 131
  information window 132
  inital file processing 107
  local files icon 132
  main graphs 109, 132
  main window 108
  metrics graphs 107, 115
  metrics selection 114
  metrics sellectin window 108
  overview 107
  pit icon 135
  print icon 136
  printing main graphs 121
  printing metrics 116
  recording files 107
  removing main graphs 120
  removing metrics from main graphs 120
  scale icon 136


© Copyright IBM Corp. 1994, 2004

azizo *(continued)*
tabular view of main graphs 120
tabular view of metrics 116
user interface 110
view icon 137
working with main graphs 117
X Resources 115
X Resources for Main Graphs 119

# C

capacity planning 2
chmon
parameters 293
syntax 293
choosing a console name 47
colors
exmon coloring scheme 90
exmon exception 90
exmon value ranges 90
for state lights 14
command line
3dmon 74
a2ptx 96
azizo 110
filtd 183
ptx2stat 105
ptxconv 100
ptxhottab 105
ptxmerge 97
ptxrlog 103
ptxsplit 98
ptxtab 101
xmperf 29
xmservd 155, 171
xmtrend 175
command menu interface
xmperf 61
commands
3dmon 290
3dplay 292
a2ptx 292
azizo 293
chmon 293
filtd 294
PTX 289
ptxconv 294
ptxmerge 294
ptxrlog 295
ptxsplit 296
ptxtab 297
wlmmon 147
wlmperf 147
xmpeek 299
xmperf 300
xmscheck 302
xmservd 303
common shared memory 154
components
agent 3
manager 4
components of Performance Toolbox for AIX 1
configuration files
3dmon 76
configurations
applying 124
deleting 125
saving 123
configuring the SpmiResp daemon 193
console
adding instruments 48
console file menu
xmperf 34
console instruments 12
console popup menus
xmperf 40
console pull-down menus
xmperf 33
console title bar 22
console windows
xmperf 33
consoles 20
adding an instrument to 21
creating 47
managing 20
moving instruments in 22
non-skeleton 20
placing instruments in 21
resizing instruments 21
skeleton 20

# D

data reduction with filtd 183
data suppliers
limiting access 164
data value properties 10
data values
a2ptx 96
data-supplier shared memory layout 212
data-suppliers
how to identify 24
when to identify 24
DDS shared memory 154
decimal places
tabulating windows 51
default instrument properties 56
default value properties 55
defining an enhanced execution of vmstat 66
defining an execution of vmstat 66
defining executables
xmperf 62
deleting configurations 125
dialog boxes
azizo 125
dialogs
important
xmperf 46
dynamic data-supplier programs 165

## E

Each skeleton type 16
edit console menu
  xmperf 35
edit value menu
  xmperf 39
environments
  configuration files 23
example
  svmon definition 64
  vmstat definition 65
example definition for renice command 69
examples
  alarm definitions 189
  filtd data reduction 186
exception identifier text
  exmon 91
exception messages
  requesting 25
executables
  defining options 63
  xmperf 62
exmon
  adding hosts 88
  color value ranges 90
  coloring scheme 90
  command execution 89
  configuration file 90
  deleting an exception log 87
  deleting hosts 88
  duplicate host names 88
  exception colors 90
  exception identifier text 91
  main window menu bar 86
  monitoring exceptions 85
  overview 85
  resource file 90
  resynchronizing hosts 88
  viewing an exception log 86
  windows
    main 85
    monitoring 85
  working with exception logs 86
  working with hosts 87

## F

file menu
  xmperf 31
files
  a2ptx input format 95
  annotation 93, 95
  azizo 282
    configuration 123
  binary recording
    ptxrlog 104
  configuration 168
    filtd 183
  exmon configuration 90
  exmon resource 90

files *(continued)*
  filtering with azizo 94
  formatting with a2ptx 95
  jazizo configuration 140
  jazizo recording 139
  merging with ptxmerge 94
  PTX 271
  recording 93
    creation 93
    modifying 94
  recording configuration 168
    command lines 171
    frequency line 169
    hot lines 172
    metric lines 171
    recovery 176
    retain line 168
    selecting metrics 173
    start-stop 170
    starting from the command line 175
    wildcards 171
  Rsi.hosts 79
  splitting with ptxsplit 94
  version conversion with ptxconv 95
  wlmperf 150
  X Resources
    3dmon 79
  xmperf 271
  xmperf configuration 272
  xmperf help file format 286
  xmperf resource 277
  xmservd 272
filtd
  alarm definition examples 189
  alarm duration and frequency 188
  alarm severity 188
  automatic start 184
  command line 183
  configuration file 183
  data reduction 184
  data reduction and alarms 183
  data reduction delay 186
  data reduction examples 186
  defining alarms 187
  overview 183
  quantities and counters 185
  rounding 187
  sampling interval 183
  termination 184
  using raw and delta values 189
  wildcards 185
frequency
  recording 139

## G

graph
  metrics
    azizo 134
guide
  performance tuning 147

## H

header lines
  xmperf tabulating window 50
help menu
  xmperf 33, 40
hierarchy
  monitoring with xmperf 9
host identifier
  a2ptx 95
hotset data
  listing recorded 105

## I

installation 268
  other than RS/6000 268
installing PTX
  prerequisites 267
    orderining information 267
Installing PTX 267
instruments 12
  adding to a console 21
  configuring 12
  hints and tips 18
  moving in consoles 22
  placing in consoles 21
  resizing in consoles 21
  skeleton 14
interface
  application programming 3
  SNMP 3

## J

jazizo 139
  configuration files 140
  hiRange
    loRange 139
  legend panel 145
  menus 140
    configuration 143
    edit 142
    file 140
    report 144
    view 143
  metric definitions 139
  metric properties 145
  recording files 139
  recording frequency 139
jtopas 262
  configuration file 263
  consoles 264
  info section 264
  menus 263
    data source 263
    file 263
    host list 263
    options 263
    reports 263
  playback panel 265
jtopas *(continued)*
  recording file 262

## K

keep metrics 120

## L

labels
  user defined 12
lines
  configuration file 168
  hot 172
  metric 171
  retain 168
  start-stop 170
localhost in xmperf 23

## M

main graphs in azizo 109
main graphs, working with azizo 117
main window
  xmperf 31
Makefiles
  remote statistics interface 245
manager 4
manager installation 268
menus
  jazizo 140
  jtopas 263
  process overview 68
messages
  exception 25
metrics 3
  counter versus quantity 140
  recording configuration file 173
  removing 117
  trend 139
metrics graph
  azizo 134
modify instrument submenu
  xmperf 36, 43
monitor menu
  xmperf 32
monitoring features 1
monitoring remote systems 153
monitoring statistics 7
multiplex interface
  SNMP 177

## N

network management principles
  SNMP 177
networked operation 2
non-skeleton consoles 20

## P

PAIDE
  see Performance Aide for AIX 1
path name display with 3dmon 74
path names 11
performance
  analyzing
    azizo 107
    jazizo 139
  monitoring with xmperf 7
performance data
  recording
    local systems 167
    remote systems 167
Performance Toolbox for AIX
  components of 1
  overview 1
playback
  xmperf 53
playback console windows
  xmperf 43
playback console, using 56
playback consoles
  creation of 55
preparser
  xmscheck 174
print box 126
print statistics, xmpeek 163
printing metrics, azizo 116
process controls
  vmstat 67
process overview
  vmstat 67
process token 69
processes
  remote 25
    list 26
    menu 27
product components 3
programs
  ptx2stat 105
  ptxconv 99
  ptxhottab 105
  ptxls 102
  ptxrlog 103
  ptxsplit 98
  recording support 93
properties
  data value 10
protocol version control 164
PTX
  see Performance Toolbox for AIX 1
PTX agent 267
PTX manager 267
ptx2stat 105
ptxconv 99
  command line 100
  version conversion 95
ptxls 102
ptxmerge 96
  command line 97
ptxmerge *(continued)*
  merging files 94
  when to use 97
ptxrlog 103
  binary recording files 104
  command line 103
  resynchronizing 105
ptxsplit 98
  command line 98
  splitting files 94
ptxtab
  command line 101
  example of default output format 100
  listing recorded data 100

## R

recording
  with 3dmon 73
  xmperf 53
recording configuration file 168
  frequency line 169
recording file inconsistencies 58
recording files 93
  azizo 107
  creation 93
  initial processing 107
recording menu
  xmperf 40
recording submenus
  xmperf 43
recovery
  recording configuration file 176
remote process list 26
remote processes 25
remote processes menu 27
Remote Statistics Interface 245, 247, 249
  adding statistics to the statset, example 252
  alternative way to decode data feeds 254
  concepts and terms 247
  data structure 247
  data-consumer decoding of data feeds 253
  data-consumer program 251
  defining a statset, example 252
  expanding the data-consumer program 254
  full-screen, character-based monitor 257
  initializing and terminating the program, example 251
  inviting data suppliers 255
  Makefiles 245
  network driven interface 249
  overview 245
  request-response interface 249
  resynchronizing 250
  subroutines 371
    defining sets of statistics to receive 246
    error codes 257
    initialization and termination 246
    instantiation and traversal of context hierarchy 246
    list 246

Remote Statistics Interface *(continued)*
subroutines *(continued)*
receiving and decoding data feed packets 247
RSiAddSetHot 371
RSiChangeFeed 374
RSiChangeHotFeed 375
RSiClose 376
RSiCreateStatSet 378
RSiDelSetHot 379
RSiDelSetStat 380
RSiFirstCx 381
RSiFirstStat 382
RSiGetHotItem 384
RSiGetRawValue 386
RSiGetValue 387
RSiInit 388
RSiInstantiate 390
RSiMainLoop 392
RSiNextCx 393
RSiNextStat 395
RSiOpen 396
RSiPathAddSetStat 398
RSiPathGetCx 400
RSiStartFeed 402
RSiStartHotFeed 403
RSiStatGetPath 401
RSiStopHotFeed 406
starting, changing and stopping data feeding 246
remote systems
monitoring
overview 153
monitoring with xmperf 23
performance data recording 167
removing metrics 117
renice
example definition 69
report box 127
report displays 149
report properties panel 148
rescan 120
response time measurement 191
application 195
ARM contexts in Spmi data space 195
ARM implementation restrictions 196
ARM library implementation 196
ARM run-time control 197
ARM Subroutines 305
ARM transaction metrics 196
configuring the SpmiResp daemon 193
introduction 191
IP 192
IP metrics 192
IP response time contexts 193
Monitoring ARM metrics from 3dmon 199
Monitoring ARM metrics from xmperf 199
monitoring IP response time from 3dmon 194
monitoring IP response time from xmperf 194
SpmiArmd daemon 197
configuring 198
resynchronizing by ptxrlog 105
resynchronizing multiple hosts with 3dmon 73
RSi
see Remote Statistic Interface 371
Rsi.hosts file 79

# S

saving configurations 123
scale icon 122
shared memory types 154
simple network management protocol
see SNMP 177
skeleton consoles 20
skeleton instruments 14
skeleton type
All 16
Each 16
SMUX
Configuration Conflicts 179
instantiation 179
instantiation rules 180
limitations induced by 179
SNMP interface 3
SNMP multiplex interface
instantiation Rules 180
interaction between xmservd and SNMP 178
limitations induced by SMUX 179
network management principles 177
overview 177
SMUX Configuration Conflicts 179
SMUX instantiation 179
socket buffer pool
adjusting 166
SPMI 153
common shared memory 154
create a dynamic data supplier 216
create main loop 224
cx_create structure
declaring a context 214
data access structures and handles 208
HotSets 209
data areas 204
data traversal structures and handles 205
data-supplier shared memory layout 212
DDS shared memory 154
declare data structures to describe contexts 221
declare data structures to describe dynamic context 228
declare data structures to describe dynamic statistics 227
declare data structures to describe statistics 220
declare other data areas as required 222
dynamic data supplier for permanent extensions 220
dynamic data supplier for volatile extensions 226
dynamic data supplier program 211
error codes, list of 242
features 202
initialize exception handling 223
initialize statistics fields 223
initialize the interface 223
instantiability 204

SPMI *(continued)*
instantiation 203
makefiles 216
making dynamic data-supplier statistics unique 214
modify main loop to add and delete dynamic context 229
modify registration with the Spmi interface 228
recognizing volatile extensions 229
releasing shared memory manually 155
shared memory data area 223
shared memory structured fields 222
shared memory types 154
SPMI data traversal program, example of 234
SPMI data user program, example of 230
SPMI dynamic data-supplier program, example of 237
SpmiCx structure 206
SpmiCxHdl handle 206
SpmiCxLink structure 207
SpmiHotItems structure 211
SpmiHotSet structure 210
SpmiHotVals structure 210
SpmiRawStat structure
declaring a statistic 213
SpmiStat structure 206
SpmiStatHdl handle 207
SpmiStatLink structure 207
SpmiStatSet structure 208
SpmiStatVals structure 209
statsets 153, 208
subroutines 241
data access 242
data hierarchy traversal 241
expand or reduce the data hierarchy 242
HotSet maintenance 241
initialize, terminate, and instantiate 241
StatSet maintenance 241
the entire program 225
traversing the data hierarchy 204
understanding the SPMI data hierarchy 202
uses 201
using the API 216
writing dynamic data-supplier programs 217
SPMI overview 201
SpmiArmd daemon
configuring 198
SpmiCx structure 206
SpmiCxHdl handle 206
SpmiCxLink structure 207
SpmiHotItems structure 211
SpmiHotSet structure 210
SpmiHotVals structure 210
SpmiStat structure 206
SpmiStatHdl handle 207
SpmiStatLink structure 207
SpmiStatSet structure 208
SpmiStatVals structure 209
state lights
colors for 14
statistic names
a2ptx 96
statistics
monitoring with xmperf 7
recording 53
trend 139
statistics, metrics, and values 3
subroutines
remote statistics interface
adding statistics to the statset, example 252
alternative way to decode data feeds 254
concepts and terms 247
data structure 247
data-consumer decoding of data feeds 253
data-consumer program 251
defining a statset, example 252
defining sets of statistics to receive 246
error codes 257
expanding the data-consumer program 254
full-screen, character-based monitor 257
initialization and termination 246
initializing and terminating the program, example 251
instantiation and traversal of context hierarchy 246
inviting data suppliers 255
list 246
network driven interface 249
receiving and decoding data feed packets 247
resynchronizing 250
RSiAddSetHot 371
RSiChangeFeed 374
RSiChangeHotFeed 375
RSiClose 376
RSiCreateStatSet 378
RSiDelSetHot 379
RSiDelSetStat 380
RSiFirstCx 381
RSiFirstStat 382
RSiGetHotItem 384
RSiGetRawValue 386
RSiGetValue 387
RSiInit 388
RSiInstantiate 390
RSiMainLoop 392
RSiNextCx 393
RSiNextStat 395
RSiOpen 396
RSiPathAddSetStat 398
RSiPathGetCx 400
RSiStartFeed 402
RSiStartHotFeed 403
RSiStatGetPath 401
RSiStopHotFeed 406
starting, changing and stopping data feeding 246
request-response interface 249
RSi 371
SPMI interface 241
data access 242
data hierarchy traversal 241
expand or reduce the data hierarchy 242
HotSet maintenance 241
initialize, terminate, and instantiate 241

subroutines *(continued)*
  SPMI interface *(continued)*
    SpmiAddSetHot 327
    SpmiCreateHotSet 330
    SpmiCreateStatSet 331
    SpmiDdsAddCx 332
    SpmiDdsDelCx 333
    SpmiDdsInit 335
    SpmiDelSetHot 336
    SpmiDelSetStat 338
    SpmiExit 339
    SpmiFirstCx 340
    SpmiFirstHot 341
    SpmiFirstStat 342
    SpmiFirstVals 343
    SpmiFreeHotSet 344
    SpmiFreeStatSet 345
    SpmiGetCx 347
    SpmiGetHotSet 348
    SpmiGetStat 349
    SpmiGetStatSet 350
    SpmiGetValue 352
    SpmiInit 353
    SpmiInstantiate 355
    SpmiNextCx 356
    SpmiNextHot 357
    SpmiNextHotItem 359
    SpmiNextStat 361
    SpmiNextVals 362
    SpmiNextValue 363
    SpmiPathAddSetStat 365
    SpmiPathGetCx 367
    SpmiStatGetPath 368
    StatSet maintenance 241
svmon
  defining an execution 65
  definition example 64
system performance measurement interface
  see SPMI 153
System Performance Measurement Interface
  see SPMI 201
  subroutines
    SpmiAddSetHot 327
    SpmiCreateHotSet 330
    SpmiCreateStatSet 331
    SpmiDdsAddCx 332
    SpmiDdsDelCx 333
    SpmiDdsInit 335
    SpmiDelSetHot 336
    SpmiDelSetStat 338
    SpmiExit 339
    SpmiFirstCx 340
    SpmiFirstHot 341
    SpmiFirstStat 342
    SpmiFirstVals 343
    SpmiFreeHotSet 344
    SpmiFreeStatSet 345
    SpmiGetCx 347
    SpmiGetHotSet 348
    SpmiGetStat 349
    SpmiGetStatSet 350
System Performance Measurement Interface *(continued)*
  subroutines *(continued)*
    SpmiGetValue 352
    SpmiInit 353
    SpmiInstantiate 355
    SpmiNextCx 356
    SpmiNextHot 357
    SpmiNextHotItem 359
    SpmiNextStat 361
    SpmiNextVals 362
    SpmiNextValue 363
    SpmiPathAddSetStat 365
    SpmiPathGetCx 367
    SpmiStatGetPath 368

## T

tabulating window decimal places 51
tabulating window title bar 51
tabulating windows
  xmperf 50
time stamps
  a2ptx 96
title bar
  consoles 22
tools menus
  xmperf 32
top monitoring
  configuration 261
Top Monitoring 261

## U

user-defined labels 12

## V

value
  changing properties 48
  path names 11
value editing submenu
  xmperf 42
value name display 17
value selection
  xmperf 46
values 3
view icon 121
vmstat
  alternative definition 66
  defining an execution 66
  definition example 65
  process controls 67
  process overview 67

## W

wildcards 14
  configuration file 171
  filtd 185

wildcards *(continued)*
- restrictions 16

WLM
- analyzing 147

WLM report browser 148
wlmmon 147
wlmperf 147
- advanced menu 149
- analysis overview 147
- daemon recording 150
- general menu 148
- tier/class menu 149

Workload Manager
- see WLM 147


# X


X Resources
- 3dmon 79

X Resources for Main Graphs, azizo 119
X Resources, azizo 115
xmpeek
- print statistics 163
- status messages 161

xmperf 29
- active recording menu items 54
- annotating while recording 59
- annotation types and fields 58
- annotations 58
- command menu interface 61
- command menus 61
- configuration file 272
- console
  - adding instruments 48
- console name
  - choosing 47
- creating a console 47
- creation of playback consoles 55
- default instrument properties 56
- default value properties 55
- defining consoles 272
- defining executables 62
- defining menus 61
- defining options for executables 63
- executables 62
- help file 286
- important dialogs 46
- instrument status 162
- introduction 7
- invoking 3dplay 84
- localhost 23
- menus
  - console file 34
  - console popup 40
  - console pull-down 33
  - edit console 35
  - edit value 39
  - file 31
  - help 33, 40
  - monitor 32
  - recording 40

xmperf *(continued)*
- menus *(continued)*
  - tools 32
- Monitoring ARM metrics from 199
- monitoring hierarchy 9
- monitoring IP response time 194
- monitoring performance 7
- monitoring remote systems with 23
- monitoring statistics with 7
- playback of recordings 55
- recording and playback 53
- recording file inconsistencies 58
- recording methods 53
- recording of statistics 53
- resource file 277
- resynchronizing in 161
- statistics 10
- submenus
  - add instrument 43
  - modify instrument 36, 43
  - recording 43
  - value editing 42
- tabulating window
  - column width 51
  - decimal places 51
  - detail lines 50
  - header lines 50
  - title bar 51
  - weighted average line 50
- user interface 29
- user interface overview 29
- using annotations 59
- using the play console 56
- value selection 46
- values 10
- windows
  - console 33
  - main 31
  - playback console 43
  - tabulating 50

xmperf files 271
xmpert
- command line 29

xmquery network protocol 159
xmscheck preparser 174
xmservd
- checking that data consumers are alive 158
- command line 155
- configuration messages 160
- data feed and data feed control messages 160
- handling exceptions 159
- interaction between xmservd and SNMP 178
- interface 157
- life and death of 157
- removing inactive data consumers 158
- rounding of sampling interval 156
- session control messages 160
- session recovery 159
- understood signals 158

xmservd files 272
xmtrend 175

# Vos remarques sur ce document / Technical publication remark form

**Titre / Title :** Bull Performance Toolbox Version 2 and 3 Guide and Reference

**Nº Reférence / Reference Nº :** 86 A2 83EM 00

**Daté / Dated :** September 2004

ERREURS DETECTEES / ERRORS IN PUBLICATION

AMELIORATIONS SUGGEREES / SUGGESTIONS FOR IMPROVEMENT TO PUBLICATION

Vos remarques et suggestions seront examinées attentivement.
Si vous désirez une réponse écrite, veuillez indiquer ci-après votre adresse postale complète.

Your comments will be promptly investigated by qualified technical personnel and action will be taken as required.
If you require a written reply, please furnish your complete mailing address below.

NOM / NAME : ______________________________ Date : __________

SOCIETE / COMPANY : ______________________________

ADRESSE / ADDRESS : ______________________________

______________________________

Remettez cet imprimé à un responsable BULL ou envoyez-le directement à :

Please give this technical publication remark form to your BULL representative or mail to:

**BULL CEDOC**
**357 AVENUE PATTON**
**B.P.20845**
**49008 ANGERS CEDEX 01**
**FRANCE**

# Technical Publications Ordering Form

## Bon de Commande de Documents Techniques

**To order additional publications, please fill up a copy of this form and send it via mail to:**
Pour commander des documents techniques, remplissez une copie de ce formulaire et envoyez-la à :


**BULL CEDOC**
**ATTN / Mr. L. CHERUBIN**
**357 AVENUE PATTON**
**B.P.20845**
**49008 ANGERS CEDEX 01**
**FRANCE**

**Phone** / Téléphone : +33 (0) 2 41 73 63 96
**FAX** / Télécopie +33 (0) 2 41 73 60 19
**E–Mail** / Courrier Electronique : srv.Cedoc@franp.bull.fr


**Or visit our web sites at:** / Ou visitez nos sites web à:

**http://www.logistics.bull.net/cedoc**

`http://www-frec.bull.com` `http://www.bull.com`

| CEDOC Reference # Nº Référence CEDOC | Qty Qté | CEDOC Reference # Nº Référence CEDOC | Qty Qté | CEDOC Reference # Nº Référence CEDOC | Qty Qté |
|---|---|---|---|---|---|
| __ __ ____ _ [__] | | __ __ ____ _ [__] | | __ __ ____ _ [__] | |
| __ __ ____ _ [__] | | __ __ ____ _ [__] | | __ __ ____ _ [__] | |
| __ __ ____ _ [__] | | __ __ ____ _ [__] | | __ __ ____ _ [__] | |
| __ __ ____ _ [__] | | __ __ ____ _ [__] | | __ __ ____ _ [__] | |
| __ __ ____ _ [__] | | __ __ ____ _ [__] | | __ __ ____ _ [__] | |
| __ __ ____ _ [__] | | __ __ ____ _ [__] | | __ __ ____ _ [__] | |
| __ __ ____ _ [__] | | __ __ ____ _ [__] | | __ __ ____ _ [__] | |
| [ _ _ ] : **no revision number means latest revision** / pas de numéro de révision signifie révision la plus récente | | | | | |

NOM / NAME : ________________________ Date : __________

SOCIETE / COMPANY : ________________________

ADRESSE / ADDRESS : ________________________

________________________

PHONE / TELEPHONE : ________________ FAX : ________________

E–MAIL : ________________________

**For Bull Subsidiaries** / Pour les Filiales Bull :

Identification: ________________________

**For Bull Affiliated Customers** / Pour les Clients Affiliés Bull :

**Customer Code** / Code Client : ________________________

**For Bull Internal Customers** / Pour les Clients Internes Bull :

**Budgetary Section** / Section Budgétaire : ________________________

**For Others** / Pour les Autres :

**Please ask your Bull representative.** / Merci de demander à votre contact Bull.

**BULL CEDOC**
**357 AVENUE PATTON**
**B.P.20845**
**49008 ANGERS CEDEX 01**
**FRANCE**

ORDER REFERENCE
86 A2 83EM 00