

# NightTrace User's Guide

---



0890398-120  
August 2004

Copyright 2004 by Concurrent Computer Corporation. All rights reserved. This publication or any part thereof is intended for use with Concurrent products by Concurrent personnel, customers, and end-users. It may not be reproduced in any form without the written permission of the publisher.

The information contained in this document is believed to be correct at the time of publication. It is subject to change without notice. Concurrent makes no warranties, expressed or implied, concerning the information contained in this document.

To report an error or comment on a specific portion of the manual, photocopy the page in question and mark the correction or comment on the copy. Mail the copy (and any additional comments) to Concurrent Computer Corporation, 2881 Gateway Drive, Pompano Beach, FL 33069-4324. Mark the envelope “**Attention: Publications Department.**” This publication may not be reproduced for any other reason in any form without written permission of the publisher.

The license management portion of this product is based on:

Élan License Manager  
Copyright 1989-1994 Elan Computer Group, Inc.  
All rights reserved.

NightTrace, KernelTrace, NightView, NightStar, Power Hawk, RedHawk, and MAXAda are trademarks of Concurrent Computer Corporation.

Élan License Manager is a trademark of Élan Computer Group, Inc.

PowerPC is a trademark of International Business Machines, Corp.

Motif, OSF, and OSF/Motif, X Window System and X are trademarks of The Open Group

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original Release -- August 1992	000	NightTrace 1.0
Current Release -- August 2004	120	NightTrace 5.4

## Scope of Manual

This manual is a reference document and users guide for NightTrace™, a graphical, interactive debugging and performance analysis tool.

## Structure of Manual

A brief description of the chapters and appendixes in this manual follows:

- Chapter 1 contains introductory material on NightTrace.
- Chapter 2 documents the NightTrace Logging API.
- Chapter 3 details the NightTrace Analysis API.
- Chapter 4 describes how to invoke the **ntrace** display utility.
- Chapter 5 introduces the NightTrace graphical user interface.
- Chapter 6 describes the **ntraceud** command line user daemon.
- Chapter 7 describes the **ntracekd** command line kernel daemon.
- Chapter 8 shows how to view trace event logs with **ntrace**.
- Chapter 9 describes display pages and their various components.
- Chapter 10 details the methods for creating and configuring display objects.
- Chapter 11 defines NightTrace expressions.
- Chapter 12 details NightTrace search and summarize capabilities.
- Chapter 13 describes kernel tracing.

This manual also contains three appendixes, a glossary, and an index.

- Appendix A describes performance tuning.
- Appendix B describes graphical user interface (GUI) customization.
- Appendix C provides answers to common questions.

The glossary contains an alphabetical list of NightTrace, X<sup>TM</sup>, and Motif<sup>TM</sup> words and phrases used in this manual and their definitions. The index contains an alphabetical list of topics, names, etc. found in the manual.

Man page descriptions of programs, system calls, subroutines, and file formats appear in the system manual pages.

## Syntax Notation

The following notation is used throughout this guide:

<i>italic</i>	Books, reference cards, and items that the user must specify appear in <i>italic</i> type. Special terms and comments in code may also appear in <i>italic</i> .
<b>list bold</b>	User input appears in <b>list bold</b> type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in <b>list bold</b> type.
list	Operating system and program output such as prompts and messages and listings of files and programs appears in list type. Keywords also appear in list type.
<u>emphasis</u>	Words or phrases that require extra emphasis use <u>emphasis</u> type.
window	Keyboard sequences and window features such as button, field, and menu labels and window titles appear in window type.
[ ]	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments.
{ }	Braces enclose mutually exclusive choices separated by the pipe ( ) character, where one choice must be selected. You do not type the braces or the pipe character with the choice.
. . .	An ellipsis follows an item that can be repeated.

## Referenced Publications

The following publications are referenced in this document:

0830048	<i>HN6200 Architecture Reference Manual</i>
0830046	<i>HN6800 Architecture Manual</i>
0890240	<i>Concurrent Fortran 77 Reference Manual</i>
0890300	<i>X Window<sup>®</sup> System User's Guide</i>

0890378	<i>C: A Reference Manual</i>
0890380	<i>OSF/Motif™ Documentation Set (3 volumes)</i>
0890395	<i>NightView™ User's Guide</i>
0890423	<i>PowerMAX OS™ Programming Guide</i>
0890429	<i>System Administration Volume 1</i>
0890430	<i>System Administration Volume 2</i>
0890460	<i>Compilation Systems Volume 2 (Concepts)</i>
0890466	<i>PowerMAX OS™ Real-Time Guide</i>
0890474	<i>NightTrace™ Pocket Reference</i>
0890516	<i>MAXAda™ Reference Manual</i>
0891019	<i>Concurrent C Reference Manual</i>
0891055	<i>Élan™ License Manager Release Notes</i>
0891082	<i>Real-Time Clock and Interrupt Module User's Guide</i>



# Contents

## Chapter 1 Introduction

What is NightTrace? .....	1-1
User and Kernel Tracing .....	1-2
Timestamp Source Selection .....	1-2
Trace-Point Placement .....	1-2
Languages Supported .....	1-3
Processes and CPUs .....	1-3
Information Displayed .....	1-3
Searches and Summaries .....	1-3
Logging and Analysis .....	1-3
The User Trace Event Logging Procedure .....	1-4
The Kernel Trace Event Logging Procedure .....	1-5
The Trace Event Analysis Procedure .....	1-6
Recommended Reading .....	1-6

## Chapter 2 Using the NightTrace Logging API

Language-Specific Source Considerations .....	2-1
C .....	2-1
Fortran .....	2-2
Ada .....	2-2
Inter-Process Communication and Library Routines .....	2-3
Understanding NightTrace Library Calls .....	2-3
trace_begin() .....	2-5
trace_open_thread() .....	2-10
trace_event() and Its Variants .....	2-11
trace_enable(), trace_disable(), and Their Variants .....	2-17
trace_flush() and trace_trigger() .....	2-21
trace_close_thread() .....	2-23
trace_end() .....	2-24
Disabling Tracing .....	2-25
Compiling and Linking .....	2-25
C Example .....	2-26
Fortran Example .....	2-26
Ada Example .....	2-26

## Chapter 3 Using the NightTrace Analysis API

NightTrace Analysis Application Programming Interface .....	3-1
Data Structures .....	3-3
tr_cb_t .....	3-3
tr_cond_cb_func_t .....	3-4
tr_cond_func_t .....	3-4
tr_cond_t .....	3-5
tr_dir_t .....	3-5

tr_offset_t . . . . .	3-5
tr_state_action_t . . . . .	3-6
tr_state_cb_func_t . . . . .	3-6
tr_state_info_t . . . . .	3-7
tr_state_t . . . . .	3-7
tr_stream_event_t . . . . .	3-8
tr_stream_func_t . . . . .	3-8
tr_string_node_t . . . . .	3-8
tr_t . . . . .	3-8
Functions . . . . .	3-9
API Initialization and Destruction . . . . .	3-13
tr_init() . . . . .	3-13
tr_destroy() . . . . .	3-13
Error Detection, Collection, and Reporting . . . . .	3-15
tr_error_clear() . . . . .	3-15
tr_error_check() . . . . .	3-16
Input Specification and Streaming Control . . . . .	3-17
tr_open_file() . . . . .	3-17
tr_open_stream() . . . . .	3-18
tr_close() . . . . .	3-19
tr_stream_notify() . . . . .	3-20
tr_stream_consume() . . . . .	3-21
tr_stream_size() . . . . .	3-21
Event Offset Positioning . . . . .	3-23
tr_next_event() . . . . .	3-23
tr_next_event_() . . . . .	3-24
tr_prev_event() . . . . .	3-24
tr_prev_event_() . . . . .	3-25
tr_search() . . . . .	3-26
tr_seek() . . . . .	3-27
Basic Event Attribute Functions . . . . .	3-28
tr_id() . . . . .	3-29
tr_id_() . . . . .	3-29
tr_time() . . . . .	3-30
tr_time_() . . . . .	3-31
tr_nargs() . . . . .	3-31
tr_nargs_() . . . . .	3-32
tr_arg_int() . . . . .	3-33
tr_arg_int_() . . . . .	3-33
tr_arg_dbl() . . . . .	3-34
tr_arg_dbl_() . . . . .	3-35
tr_pid() . . . . .	3-35
tr_pid_() . . . . .	3-36
tr_raw_pid() . . . . .	3-38
tr_raw_pid_() . . . . .	3-39
tr_lwpid() . . . . .	3-40
tr_lwpid_() . . . . .	3-41
tr_tid() . . . . .	3-41
tr_tid_() . . . . .	3-42
tr_thread_id() . . . . .	3-43
tr_thread_id_() . . . . .	3-43
tr_task_id() . . . . .	3-44
tr_task_id_() . . . . .	3-45
tr_cpu() . . . . .	3-45



tr_cpu_()	3-46
tr_node()	3-47
tr_node_()	3-48
tr_process_name()	3-48
tr_process_name_()	3-49
tr_task_name()	3-50
tr_task_name_()	3-50
tr_thread_name()	3-51
tr_thread_name_()	3-51
Conditions	3-53
tr_cond_create()	3-54
tr_cond_reset()	3-55
tr_cond_find()	3-55
tr_cond_id()	3-56
tr_cond_id_range()	3-57
tr_cond_id_clear()	3-58
tr_cond_cpu()	3-59
tr_cond_cpu_clear()	3-59
tr_cond_pid()	3-60
tr_cond_pid_name()	3-61
tr_cond_pid_clear()	3-62
tr_cond_tid()	3-63
tr_cond_tid_name()	3-64
tr_cond_tid_clear()	3-65
tr_cond_node()	3-66
tr_cond_node_clear()	3-67
tr_cond_func_or()	3-68
tr_cond_func_and()	3-70
tr_cond_func_clear()	3-72
tr_cond_expr_and()	3-73
tr_cond_expr_or()	3-74
tr_cond_not()	3-75
tr_cond_or()	3-76
tr_cond_and()	3-77
tr_cond_copy()	3-78
tr_cond_name()	3-79
tr_cond_satisfy()	3-79
tr_cond_satisfy_()	3-80
tr_cond_register()	3-81
tr_cond_offset()	3-82
State-oriented Interfaces	3-83
tr_state_create()	3-84
tr_state_find()	3-85
tr_state_name()	3-85
tr_state_start_id()	3-86
tr_state_start_id_range()	3-87
tr_state_start_id_clear()	3-88
tr_state_end_id()	3-88
tr_state_end_id_range()	3-89
tr_state_end_id_clear()	3-90
tr_state_start_cond()	3-90
tr_state_start_cond_clear()	3-91
tr_state_end_cond()	3-92
tr_state_end_cond_clear()	3-92

tr_activate()	3-93
tr_state_info()	3-94
tr_state_info_()	3-95
tr_state_active()	3-96
tr_state_active_()	3-97
Output Function	3-98
tr_copy_input()	3-98
String Table Functions	3-99
tr_get_string()	3-99
tr_get_item()	3-100
tr_create_table()	3-101
tr_append_table()	3-102
Callback Interfaces	3-103
tr_iterate()	3-103
tr_halt()	3-104
tr_cancel_cb()	3-104
tr_cond_cb()	3-105
tr_state_cb()	3-106
Sample Programs	3-107
list	3-108
list.c	3-108
search	3-110
search.c	3-110
watchdog	3-112
watchdog.c	3-112
ptime	3-115
ptime.c	3-116
browse	3-118
browse.c	3-118

## Chapter 4 Invoking NightTrace

Command-line Options	4-1
Summary Criteria	4-5
Command-line Arguments	4-9
Trace Event Files	4-10
Event Map Files	4-10
Page Configuration Files	4-13
Tables	4-13
String Tables	4-15
Pre-Defined String s	4-16
Format Tables	4-19
Pre-Defined Format Tables	4-23
Session Configuration Files	4-24
Trace Data Segments	4-25

## Chapter 5 Using the NightTrace GUI

NightTrace Main Window Menu Bar	5-3
NightTrace	5-3
Unsaved Changes	5-7
Daemons	5-10
Login	5-14

Enter Password . . . . .	5-14
Attach Daemons . . . . .	5-16
Pages . . . . .	5-18
Build Custom Kernel Page . . . . .	5-21
Select Graphs . . . . .	5-23
Options . . . . .	5-25
Refresh Interval . . . . .	5-26
Display Buffer Size Warning . . . . .	5-27
Tools . . . . .	5-28
Help . . . . .	5-30
Session Configuration File Name Area . . . . .	5-31
Daemon Control Area . . . . .	5-32
Enable / Disable Trace Events . . . . .	5-37
Session Overview Area . . . . .	5-39
Daemon Definition Dialog . . . . .	5-41
Import Daemon Definition . . . . .	5-44
General . . . . .	5-46
Target . . . . .	5-46
Trace Events Output . . . . .	5-48
User Trace . . . . .	5-51
Locking Policies . . . . .	5-51
Shared Memory . . . . .	5-53
Timestamp Heartbeat . . . . .	5-53
User Event Buffer . . . . .	5-53
Inheritance . . . . .	5-54
Events . . . . .	5-55
Runtime . . . . .	5-57
Scheduling . . . . .	5-57
CPU Bias . . . . .	5-58
NUMA . . . . .	5-59
Policies . . . . .	5-60
Other . . . . .	5-61
Streaming Options . . . . .	5-61
Kernel Trace Buffer Options . . . . .	5-62

## Chapter 6 Generating Trace Event Logs with ntraceud

The ntraceud Daemon . . . . .	6-1
The Default User Daemon Configuration . . . . .	6-2
ntraceud Modes . . . . .	6-4
ntraceud Options . . . . .	6-5
Option to Get Help (-help) . . . . .	6-7
Option to Get Version Information (-version) . . . . .	6-8
Option to Disable the IPL Register (-ipldisable) . . . . .	6-9
Option to Prevent Page Locking (-lockdisable) . . . . .	6-11
Option to Establish File-Wraparound Mode (-filewrap) . . . . .	6-12
Option to Establish Buffer-Wraparound Mode (-bufferwrap) . . . . .	6-13
Option to Define Shared Memory Buffer Size (-memsize) . . . . .	6-16
Option to Set Timeout Interval (-timeout) . . . . .	6-17
Option to Set the Buffer-Full Cutoff Percentage (-cutoff) . . . . .	6-18
Option to Select Timestamp Source (-clock) . . . . .	6-19
Option to Reset the ntraceud Daemon (-reset) . . . . .	6-20
Option to Quit Running ntraceud (-quit) . . . . .	6-21

- Option to Present Statistical Information (-stats) . . . . . 6-22
- Option to Disable Logging (-disable) . . . . . 6-24
- Option to Enable Logging (-enable) . . . . . 6-26
- Invoking ntraceud . . . . . 6-28

**Chapter 7 Generating Trace Event Logs with ntracekd**

- The ntracekd Daemon . . . . . 7-1
- ntracekd Modes . . . . . 7-1
- ntracekd Options . . . . . 7-2
- ntracekd Invocations . . . . . 7-4

**Chapter 8 Viewing Trace Event Logs**

- Mouse Button Operations . . . . . 8-2
- Viewing Strategy . . . . . 8-2
  - Editing Single Fields . . . . . 8-4
  - Editing Multiple Fields . . . . . 8-4

**Chapter 9 Display Pages**

- Default Display Page . . . . . 9-1
- Menu Bar . . . . . 9-3
  - Page . . . . . 9-3
  - Edit . . . . . 9-5
    - Tags . . . . . 9-7
    - Edit String Tables . . . . . 9-9
      - Edit String Table . . . . . 9-13
        - Edit String Table Entry . . . . . 9-15
        - Edit Event Map Entry . . . . . 9-16
    - Create . . . . . 9-19
    - Actions . . . . . 9-21
    - Options . . . . . 9-24
    - Help . . . . . 9-26
  - Message Display Area . . . . . 9-28
  - Grid . . . . . 9-28
  - Interval Scroll Bar . . . . . 9-30
  - Interval Push Buttons . . . . . 9-31
  - Mode Button . . . . . 9-36
  - Interval Control Area . . . . . 9-37

**Chapter 10 Display Objects**

- Types of Display Objects . . . . . 10-3
  - Grid Label . . . . . 10-4
  - Data Box . . . . . 10-5
  - Column . . . . . 10-6
  - Event Graph . . . . . 10-6
  - State Graph . . . . . 10-7
  - Data Graph . . . . . 10-8
  - Ruler . . . . . 10-10
- Operations on Display Objects . . . . . 10-12

Creating Display Objects	10-12
Selecting Display Objects	10-13
Moving Display Objects	10-14
Resizing Display Objects	10-14
Configuring Display Objects	10-15
Grid Label	10-17
Data Box	10-19
Event Graph	10-26
State Graph	10-32
Data Graph	10-39
Ruler	10-47
Configuration Dialog Push Buttons	10-48
Common Configuration Parameters	10-50
Font	10-50
Events	10-50
Condition	10-51
Processes	10-51
Threads	10-52
Nodes	10-53
Horizontal Alignment	10-53
Vertical Alignment	10-54
Color Selection	10-54
Font Selection	10-55

## Chapter 11 Using Expressions

Expressions	11-1
Operators	11-1
Operands	11-2
Constants	11-2
Functions	11-4
Function Parameters	11-7
Function Terminology	11-8
Trace Event Functions	11-14
id()	11-15
arg()	11-16
arg_dbl()	11-17
num_args()	11-18
pid()	11-19
raw_pid()	11-20
lwpid()	11-21
thread_id()	11-22
task_id()	11-23
tid()	11-24
cpu()	11-25
offset()	11-26
time()	11-27
node_id()	11-28
pid_table_name()	11-29
tid_table_name()	11-30
node_name()	11-31
process_name()	11-32
task_name()	11-33

thread_name()	11-34
Multi-Event Functions	11-35
event_gap()	11-35
event_matches()	11-36
State Functions	11-37
Start Functions	11-37
start_id()	11-38
start_arg()	11-39
start_arg_dbl()	11-40
start_num_args()	11-41
start_pid()	11-42
start_raw_pid()	11-43
start_lwpid()	11-44
start_thread_id()	11-45
start_task_id()	11-46
start_tid()	11-47
start_cpu()	11-48
start_offset()	11-49
start_time()	11-50
start_node_id()	11-51
start_pid_table_name()	11-52
start_tid_table_name()	11-53
start_node_name()	11-54
End Functions	11-55
end_id()	11-56
end_arg()	11-57
end_arg_dbl()	11-58
end_num_args()	11-59
end_pid()	11-60
end_raw_pid()	11-61
end_lwpid()	11-62
end_thread_id()	11-63
end_task_id()	11-64
end_tid()	11-65
end_cpu()	11-66
end_offset()	11-67
end_time()	11-68
end_node_id()	11-69
end_pid_table_name()	11-70
end_tid_table_name()	11-71
end_node_name()	11-72
Multi-State Functions	11-73
state_gap()	11-73
state_dur()	11-74
state_matches()	11-75
state_status()	11-76
Offset Functions	11-77
offset_id()	11-78
offset_arg()	11-79
offset_arg_dbl()	11-80
offset_num_args()	11-81
offset_pid()	11-82
offset_raw_pid()	11-83
offset_lwpid()	11-84

offset_thread_id()	11-85
offset_task_id()	11-86
offset_tid()	11-87
offset_cpu()	11-88
offset_time()	11-89
offset_node_id()	11-90
offset_pid_table_name()	11-91
offset_tid_table_name()	11-92
offset_node_name()	11-93
offset_process_name()	11-94
offset_task_name()	11-95
offset_thread_name()	11-96
Summary Functions	11-97
min()	11-97
max()	11-98
avg()	11-99
sum()	11-100
min_offset()	11-101
max_offset()	11-102
summary_matches()	11-103
Format and Table Functions	11-104
get_string()	11-104
get_item()	11-106
get_format()	11-108
format()	11-110
Macros	11-111
Qualified Events	11-113
Qualified States	11-116
NightTrace Qualified Expressions	11-119
Edit NightTrace Qualified Expression	11-122

## Chapter 12 Search and Summarize

Searching for Points of Interest	12-1
Search Options	12-10
Summarizing Statistical Information	12-12
Criteria	12-14
Options	12-26

## Chapter 13 Tracing the Kernel

Default Kernel Trace Points	13-1
Context Switch Trace Event	13-2
Interrupt Trace Events	13-2
Exception Trace Events	13-3
Syscall Trace Events	13-4
Kernel Trace Points Not Enabled By Default	13-5
Page Fault Event	13-5
Protection Fault Event	13-5
Viewing Kernel Trace Event Files	13-6
Kernel Display Pages	13-6
Node and CPU Information	13-7
Running Process Information	13-8

Node Information . . . . .	13-8
Context Switch Information . . . . .	13-9
Interrupt Information . . . . .	13-9
Exception Information . . . . .	13-10
Syscall Information . . . . .	13-12
Color Information . . . . .	13-13
Kernel String Tables . . . . .	13-13
Kernel Reference . . . . .	13-15
Interrupts . . . . .	13-15
Non-Device-Related Interrupts . . . . .	13-16
Device-Related Interrupts . . . . .	13-16
Exceptions . . . . .	13-17
Syscalls . . . . .	13-18

## Appendix A Performance Tuning

Preventing Trace Events Loss . . . . .	A-1
Ensuring Accurate Timings . . . . .	A-3
Optimizing File System and CPU Usage . . . . .	A-3
Conserving Disk Space . . . . .	A-4
Conserving Memory and Accelerating ntrace . . . . .	A-4

## Appendix B GUI Customization

Default X-Resource Settings for ntrace . . . . .	B-2
Examples . . . . .	B-3
Exercise: Customizing Display Colors . . . . .	B-3

## Appendix C Answers to Common Questions

### Illustrations

Figure 1-1. Example of Instrumented C Code . . . . .	1-4
Figure 2-1. Inter-Process Communication and Library Routines . . . . .	2-4
Figure 5-1. NightTrace Main Window . . . . .	5-1
Figure 5-2. NightTrace menu . . . . .	5-3
Figure 5-3. Unsaved Changes / Exit dialog . . . . .	5-7
Figure 5-4. Unsaved Changes / Proceed dialog . . . . .	5-8
Figure 5-5. Daemons menu . . . . .	5-10
Figure 5-6. Login dialog . . . . .	5-14
Figure 5-7. Enter Password dialog . . . . .	5-15
Figure 5-8. Attach Daemons dialog . . . . .	5-16
Figure 5-9. Pages menu . . . . .	5-18
Figure 5-10. New Display Page . . . . .	5-19
Figure 5-11. Build Custom Kernel Page dialog . . . . .	5-21
Figure 5-12. Select CPUs dialog . . . . .	5-21
Figure 5-13. Select Graphs dialog . . . . .	5-23
Figure 5-14. Options menu . . . . .	5-25
Figure 5-15. Refresh Interval dialog . . . . .	5-26
Figure 5-16. Display Buffer Size Warning dialog . . . . .	5-27
Figure 5-17. Tools menu . . . . .	5-28
Figure 5-18. Help menu . . . . .	5-30



Figure 5-19. Session Configuration File Name Area ..... 5-31

Figure 5-20. Daemon Control Area ..... 5-32

Figure 5-21. Enable / Disable Trace Events dialog ..... 5-37

Figure 5-22. Session Overview Area ..... 5-39

Figure 5-23. Daemon Definition dialog ..... 5-41

Figure 5-24. Import Daemon Definition dialog ..... 5-44

Figure 5-25. Daemon Definition dialog - General ..... 5-46

Figure 5-26. Daemon Definition dialog - User Trace ..... 5-51

Figure 5-27. Daemon Definition dialog - Events ..... 5-55

Figure 5-28. Daemon Definition dialog - Runtime ..... 5-57

Figure 5-29. Daemon Definition dialog - Other ..... 5-61

Figure 8-1. Deciding What to Do Next in View Mode ..... 8-3

Figure 9-1. A Default Display Page ..... 9-2

Figure 9-2. Display Page - Page menu ..... 9-3

Figure 9-3. Display Page - Edit menu ..... 9-5

Figure 9-4. Tags dialog ..... 9-7

Figure 9-5. Set Tag Name dialog ..... 9-9

Figure 9-6. Edit String Tables dialog ..... 9-10

Figure 9-7. Add Table dialog ..... 9-10

Figure 9-8. Save Selected String Tables dialog ..... 9-12

Figure 9-9. Edit String Table dialog ..... 9-13

Figure 9-10. Edit String Table Entry dialog ..... 9-15

Figure 9-11. Variable Argument dialog ..... 9-16

Figure 9-12. Edit Event Map Entry dialog ..... 9-17

Figure 9-13. Display Page - Create menu ..... 9-19

Figure 9-14. Display Page - Actions menu ..... 9-21

Figure 9-15. Display Page - Options menu ..... 9-24

Figure 9-16. Amount of Scrolling Due to Increment Value ..... 9-25

Figure 9-17. Display Page - Help menu ..... 9-26

Figure 9-18. Message Display Area ..... 9-28

Figure 9-19. The Grid ..... 9-29

Figure 9-20. The Interval Scroll Bar ..... 9-30

Figure 9-21. Interval Push Buttons ..... 9-31

Figure 9-22. Create New Tag dialog ..... 9-32

Figure 9-23. Discard Trace Events dialog ..... 9-34

Figure 9-24. Mode Button ..... 9-36

Figure 9-25. Interval Control Area ..... 9-37

Figure 10-1. Grid Label Examples ..... 10-4

Figure 10-2. Data Box Examples ..... 10-5

Figure 10-3. Column Example ..... 10-6

Figure 10-4. Event Graph Example ..... 10-7

Figure 10-5. State Graph Example ..... 10-7

Figure 10-6. Data Graph Examples ..... 10-9

Figure 10-7. Ruler Example ..... 10-10

Figure 10-8. Ruler Indicators ..... 10-11

Figure 10-9. Configure Grid Label dialog ..... 10-17

Figure 10-10. Configure Data Box dialog ..... 10-19

Figure 10-11. Configure Event Graph dialog ..... 10-26

Figure 10-12. Configure State Graph dialog ..... 10-32

Figure 10-13. Configure Data Graph dialog ..... 10-39

Figure 10-14. Fill Style - Solid vs. None ..... 10-45

Figure 10-15. Maximum vs. Minimum Values ..... 10-46

Figure 10-16. Configure Ruler dialog ..... 10-47

Figure 10-17. Mark and Tag Indicators ..... 10-48

Figure 10-18. Horizontal Alignment	10-53
Figure 10-19. Vertical Alignment	10-54
Figure 10-20. Choose Color dialog	10-55
Figure 10-21. Choose Font dialog	10-55
Figure 11-1. Function Terminology Illustrated	11-9
Figure 11-2. States and Events	11-10
Figure 11-3. Choosing a Key / Value pair for a qualified event	11-114
Figure 11-4. Customizing a qualified state	11-117
Figure 11-5. NightTrace Qualified Expressions dialog	11-119
Figure 11-6. Edit NightTrace Qualified Expression dialog	11-122
Figure 12-1. Search NightTrace Events dialog	12-1
Figure 12-2. Search Options dialog	12-10
Figure 12-3. Summary results displayed in Summarize dialog	12-12
Figure 12-4. Summarize NightTrace Events dialog - Criteria page	12-14
Figure 12-5. State Summary Graph	12-24
Figure 12-6. Summarize NightTrace Events dialog - Options page	12-26
Figure 13-1. Sample Kernel Display Page	13-6
Figure 13-2. Per-CPU Information	13-7
Figure 13-3. Node and CPU Box	13-7
Figure 13-4. Running Process Boxes	13-8
Figure 13-5. Node Box	13-8
Figure 13-6. Context Switch Lines	13-9
Figure 13-7. Last Interrupt Box and Interrupt Graph	13-9
Figure 13-8. Last Exception Box and Exception Graph	13-10
Figure 13-9. TR_PAGEFLT_ADDR and TR_PROTFLT_ADDR Events	13-11
Figure 13-10. TR_SWITCHIN vs. TR_PAGEFLT_ADDR and TR_PROTFLT_ADDR Events	13-12
Figure 13-11. Last Syscall Box and Syscall Graph	13-12
Figure 13-12. Color Key	13-13

**Screens**

Screen 6-1. Sample Output from the ntraceud -help Option	6-7
Screen 6-2. Sample Output from ntraceud -stats Option	6-23

**Tables**

Table 6-1. NightTrace Configuration Defaults	6-4
Table 6-2. Mode-Selection Guidelines	6-5
Table 6-3. NightTrace Operating Modes	6-5
Table 6-4. ntraceud Disable Sequence #1	6-25
Table 6-5. ntraceud Disable Sequence #2	6-25
Table 6-6. ntraceud Enable Sequence #1	6-27
Table 6-7. ntraceud Enable Sequence #2	6-27
Table 8-1. View-Mode Mouse Button Operations	8-2
Table 8-2. Valid Multiple Field Changes	8-5
Table 9-1. Manipulating the Interval Scroll Bar	9-31
Table 10-1. Examples of Conditions	10-51
Table 11-1. Time Units and Constant Suffixes	11-3
Table 11-2. NightTrace Functions	11-5
Table 13-1. Example Logical CPU Mapping	13-8
Table 13-2. Non-Device-Related Interrupt Reference	13-16
Table 13-3. Device-Related Interrupt Reference	13-17

Table 13-4. Exception Reference . . . . . 13-18  
Table B-1. Suggested Colors for X Resources . . . . . B-3

**Glossary**

**Index**



---

What is NightTrace? .....	1-1
User and Kernel Tracing .....	1-2
Timestamp Source Selection .....	1-2
Trace-Point Placement .....	1-2
Languages Supported .....	1-3
Processes and CPUs .....	1-3
Information Displayed .....	1-3
Searches and Summaries .....	1-3
Logging and Analysis .....	1-3
The User Trace Event Logging Procedure .....	1-4
The Kernel Trace Event Logging Procedure .....	1-5
The Trace Event Analysis Procedure .....	1-6
Recommended Reading .....	1-6



This chapter provides an overview of NightTrace, steps involved in using the toolset, and recommended readings.

## What is NightTrace?

NightTrace is part of the NightStar™ family and consists of an interactive debugging and performance analysis tool, trace data collection daemons, and two Application Programming Interfaces (APIs) allowing user applications to log data values as well as analyze data collected from user or kernel daemons. NightTrace allows you to graphically display information about important events in your application and the kernel, for example, event occurrences, timings, and data values. NightTrace consists of the following parts:

<b>ntrace</b>	a graphical tool that controls daemon sessions and displays user and kernel trace events in trace event file(s)
<b>ntraceud</b>	a daemon program that copies user applications' trace events from shared memory to trace event file(s)
<b>ntracekd</b>	a daemon program that copies operating system kernel trace events from kernel memory to trace event file(s)
NightTrace Logging API	libraries and include files for use in user applications that log trace events to shared memory
NightTrace Analysis API	libraries and include files for use in user applications that want to analyze data collected from user or kernel daemons

NightTrace is flexible. As a user, you control:

- Selection of user tracing of your application or kernel tracing
- Selection of timestamp source
- Trace-point placement within your application
- The source language of the trace application
- The number of processes and CPUs you gather data on
- The amounts and types of information you display

- Trace event searches and summaries

## User and Kernel Tracing

If interactions are important, you can simultaneously capture event information from your application and from the kernel. Alternatively, you can capture just user events or pre-defined kernel events.

## Timestamp Source Selection

By default, an architecture-specific timing source is utilized. For Intel-based machines, the Intel Time Stamp Counter (TSC register); for Night Hawk 6000 series machine, the interval timer; for PowerHawk and PowerStack series machines, the Time Base Register (TBR). However, the Real-Time Clock and Interrupt Module (RCIM) can be also used as a timestamp source.

The RCIM is a hardware module which provides a variety of clocks and interrupts sources, including two high-resolution timers which may be synchronized between multiple systems. Use of the RCIM timing source by NightTrace is advantageous when gathering data from multiple systems simultaneously. NightTrace can then present a synchronized view of user and kernel activity on multiple systems from a single session.

For more information about the RCIM, please see the `clock_synchronize(1M)`, `rcim(7)`, `rcimconfig(1M)`, and `sync_clock(7)` man pages.

## Trace-Point Placement

A *trace point* is a place of interest in the source code. At each user trace point, you make your application log some user-specified information along with a timestamp and some additional system information. This logged information is collectively called a *trace event*. In user traces, each trace event has a user-defined *trace event ID* number, and two different trace event IDs delimit the boundaries of a user-defined *state*.

Some typical user trace-point locations include:

- Suspected bug locations
- Process, subprogram, or loop entry and exit points
- Timing points, especially for clocking I/O processing
- Synchronization points/multi-process interaction
- Endpoints of atomic operations
- Endpoints of shared memory access code

Careful trace point placement allows you to easily identify patterns and anomalies in your application's behavior.



Kernel trace points and trace events are pre-defined and embedded in the kernel source.

## Languages Supported

On PowerMAX OS™ systems, Concurrent C, C++, Fortran, and Ada compilers are supported.

On RedHawk™ Linux® systems, Concurrent Ada and Fortran compilers are supported, as well as GNU C, C++, and Fortran.

## Processes and CPUs

A user daemon is responsible for actually recording the trace events logged by an application to disk. It can interact with single-process and multi-process applications; the processes may even run on different CPUs. When you log a trace event, NightTrace identifies both the originating process and optionally the CPU. User daemons are initiated and managed via the **ntrace** graphical tool or via the **ntraceud** command line tool.

## Information Displayed

The **ntrace** display utility lets you examine some or all trace events. Data appear as numerical statistics and as graphical images. You can create and configure the graphical components called *display objects* or use the defaults. By creating your own display objects, you can make the graphical displays more meaningful to you. You can customize display objects to reflect your preferences in content, labeling, position, size, color, and font.

## Searches and Summaries

With the **ntrace** display utility, you can perform searches and summaries. Searches let you filter out unwanted data and zero-in on trouble spots and specific data. Summaries let you define characteristics of the trace event data to be summarized in several different ways.

## Logging and Analysis

NightTrace supports two activities: trace event logging and trace event analysis.

## The User Trace Event Logging Procedure

The following text describes user trace event logging. Follow these steps in the order shown:

1. Establish a suitable environment for running your application and capturing trace data. Make sure you meet all the system requirements discussed in the *NightTrace Release Notes* for the version you are running.
2. Select trace points in your source code. A trace point marks a point in your application that is important to debugging or performance analysis.
3. Insert a call to a NightTrace trace event logging routine at each trace point in your source code, so you can later see the trace event information graphically in **ntrace**. You can manually insert these calls into your source code or insert them into the final executable with the NightView debugger. See the *NightView User's Guide* for more information.
4. Insert calls at appropriate places in your application to initialize the NightTrace trace event logging library and terminate logging. This is necessary for resource allocation and deallocation, file creation, and flushing trace events to disk. Steps 3 and 4 are called *instrumenting your code*. Figure 1-1 shows instrumented C code.

```
#include <ntrace.h>
#define START 10
#define END 20

main()
{
    trace_begin( "log", 0 );
    trace_open_thread( "main_thread" );
    trace_event( START );

    process();

    trace_event( END );
    trace_close_thread();
    trace_end();
    exit( 0 );
}
```

**Figure 1-1. Example of Instrumented C Code**

5. Compile and link your application with the NightTrace trace event logging library. For example:

```
$ cc main.c process.c -lntrace -lud # for PowerMAX
$ cc main.c process.c -lntrace -lcur_rt # for RedHawk
```

6. Start NightTrace and use it to define a daemon session used to capture user and/or kernel data. For example:

```
$ ntrace &
```

In the NightTrace Main Manager window, select the **Daemon -> New** menu item which brings up a **Daemons Definition** dialog. Click on the **User Application** radio button to define this as a user daemon. Click on the **Stream** checkbox to ensure it is now unchecked. Enter the filename passed to the `trace_begin()` routine in the text field for the **Key File**. Click the **OK** button.

7. Start the user daemon by clicking on the **Start** button in the Main window. Once the state for the daemon changes to **Paused**, click the **Resume** button.
8. Run your application. As NightTrace trace event logging routines execute, they write trace event information into a shared memory buffer. Periodically, the user daemon copies this information to a trace event file on disk. For example:

```
$ a.out
```

9. When the application completes, or when you have captured sufficient data that you now wish to analyze, stop the daemon by pressing the **Flush** button followed by the **Stop** button.
10. To display the data, press the **Display** button.

## The Kernel Trace Event Logging Procedure

Alternatively, to log and view kernel data, invoke the **ntrace** command and follow these steps:

```
$ ntrace &
```

1. Define a kernel daemon in the NightTrace Main window by selecting the **Daemon -> New** menu item which brings up a **Daemons Definition** dialog. Click on the **Kernel** radio button to define this as a kernel daemon. Click on the **Stream** checkbox to ensure it is now unchecked. Enter an output filename, such as **/tmp/kernel-data** in the text field for **Output File**. Click the **OK** button.
2. Start the kernel daemon by clicking on the **Start** button in the NightTrace Main window. Once the state displayed in the **Daemon Control** area for the daemon changes to **Paused**, click the **Resume** button.
3. Allow the daemon to capture data for a few seconds, then click on the **Flush** button followed by the **Stop** button.
4. To display the kernel data, click on the **Display** button. This will cause a default kernel page to pop up. Repeatedly click on the **Zoom Out** button on that page until you see data in the display pane. Note: if any display page is already open, clicking the **Display** button will not automatically

create a kernel display page. In such a case, open a default kernel page from the main NightTrace dialog.

## The Trace Event Analysis Procedure

1. Iteratively locate and analyze significant data.
  - Search for trace events of interest. You do this by controlling the window that displays a portion of the trace event file. This window is called the *interval*. You can control the interval by zooming in or out, scrolling, searching for specific trace events, or jumping to portions of the trace event file.
  - Display summary information. This information may be about your entire trace session or the characteristics of particular trace events and states in this trace session.

## Recommended Reading

Referenced publications appear in the front of this manual. Related text books that are useful resources for general background information follow.

### *X Window System User's Guide*

This text book by Valerie Quercia and Tim O'Reilly is published by O'Reilly & Associates, Inc. It is available under publication number 0890300. This text book introduces X terminology and concepts. It also discusses several popular window managers, the **xterm** terminal emulator, X resources, and X desk accessories.

### *OSF/Motif Style Guide*

This text book is published by Prentice-Hall, Inc. It and its companion books *OSF/Motif User's Guide* and *OSF/Motif Programmer's Guide* are packaged together under publication number 0890380. This text book introduces Motif terminology and concepts. It also provides information about Motif features.

# Using the NightTrace Logging API

---

Language-Specific Source Considerations . . . . .	2-1
C . . . . .	2-1
Fortran . . . . .	2-2
Ada . . . . .	2-2
Inter-Process Communication and Library Routines . . . . .	2-3
Understanding NightTrace Library Calls . . . . .	2-3
trace_begin() . . . . .	2-5
trace_open_thread() . . . . .	2-10
trace_event() and Its Variants . . . . .	2-11
trace_enable(), trace_disable(), and Their Variants . . . . .	2-17
trace_flush() and trace_trigger() . . . . .	2-21
trace_close_thread() . . . . .	2-23
trace_end() . . . . .	2-24
Disabling Tracing . . . . .	2-25
Compiling and Linking . . . . .	2-25
C Example . . . . .	2-26
Fortran Example . . . . .	2-26
Ada Example . . . . .	2-26



## Using the NightTrace Logging API

---

This chapter describes language-specific considerations for using NightTrace with user applications.

### CAUTION

Do not call `clock_gettime()` from your application. This system call can corrupt both the system interval timer and Time Base Register which NightTrace uses for trace event timings.

## Language-Specific Source Considerations

NightTrace applications must be written in C, Fortran, or Ada.

On RedHawk Linux, the NightTrace Logging API can be used with the following compilers:

- Concurrent Ada (MAXAda)
- Concurrent Fortran 77
- GNU C/C++
- GNU Fortran

On PowerMAX OS, the NightTrace Logging API can be used with the following compilers:

- Concurrent Ada (MAXAda)
- Concurrent C/C++
- Concurrent Fortran 77

For your applications to trace events, you must edit your source code and insert NightTrace library routine calls (unless you are using the NightView debugger). This is called *instrumenting your code*. Before you begin this task, read the following section that applies to the language in which your application is written.

## C

NightTrace applications written in C include the NightTrace header file `/usr/include/ntrace.h` with the following line:

```
#include <ntrace.h>
```

The **ntrace.h** file contains the following:

- Function prototypes for all NightTrace library routines
- Return values for all NightTrace library routines
- C macros (described in “Disabling Tracing” on page 2-25)

The library routine return values identify the type of error, if any, the NightTrace routine encountered. If you think you may want to disable the NightTrace library routines in the future without having to remove them from your source code, then you must include this file in your application.

C programs that are multi-thread can also be traced with the NightTrace library routines. For multi-thread programs, a C thread identifier is stored in each trace event, uniquely identifying which C thread was running at the time the trace event was logged.

## Fortran

All NightTrace library routines return `INTEGERS`, but because they begin with a “t”, Fortran implicitly types them as `REAL`. You must explicitly type them as `INTEGER` so that they work correctly. For example, to explicitly type the `trace_begin` routine, use the following declaration:

```
integer trace_begin
```

## Ada

Ada applications can access the NightTrace library routines via the Ada package `night_trace_bindings` which is included with the `MAXAda` product (currently only available on PowerMAX OS systems). The bindings can be found in the **bindings/general** environment in the source file **night\_trace.a**.

The `night_trace_bindings` package contains the following:

- An enumeration type consisting of the return values for all NightTrace library routines
- The bindings that permit Ada applications to call the C routines in the NightTrace library and to link in the NightTrace library

Many of the NightTrace functions have been overloaded as procedures. These procedures act as the corresponding functions, except they discard any error return values.

Ada programs that use tasking can also be traced with the NightTrace library routines. For multitasking programs, an Ada task identifier is stored in each trace event, uniquely identifying which Ada task was running at the time the trace event was logged.

For more information on Ada, see the section titled “NightTrace Binding” in the *MAXAda Reference Manual*.



## Inter-Process Communication and Library Routines

Your application logs trace events to the shared memory buffer. Later, a user daemon copies trace events from the shared memory buffer to the trace event file. The relationship between your application and the user daemon and the sequence of library calls needed to maintain this relationship appears in Figure 2-1.

## Understanding NightTrace Library Calls

There is a C, Fortran, and Ada version of each NightTrace library routine. These routines perform the following functions:

- Initialize a trace
- Open the current thread for trace event logging
- Log trace events to shared memory
- Enable and disable specified trace events
- Copy trace events from shared memory to disk
- Close the current thread for trace event logging
- Terminate a trace

See the *NightTrace Pocket Reference* card for a syntax summary of these routines. The next sections describe these routines in detail.

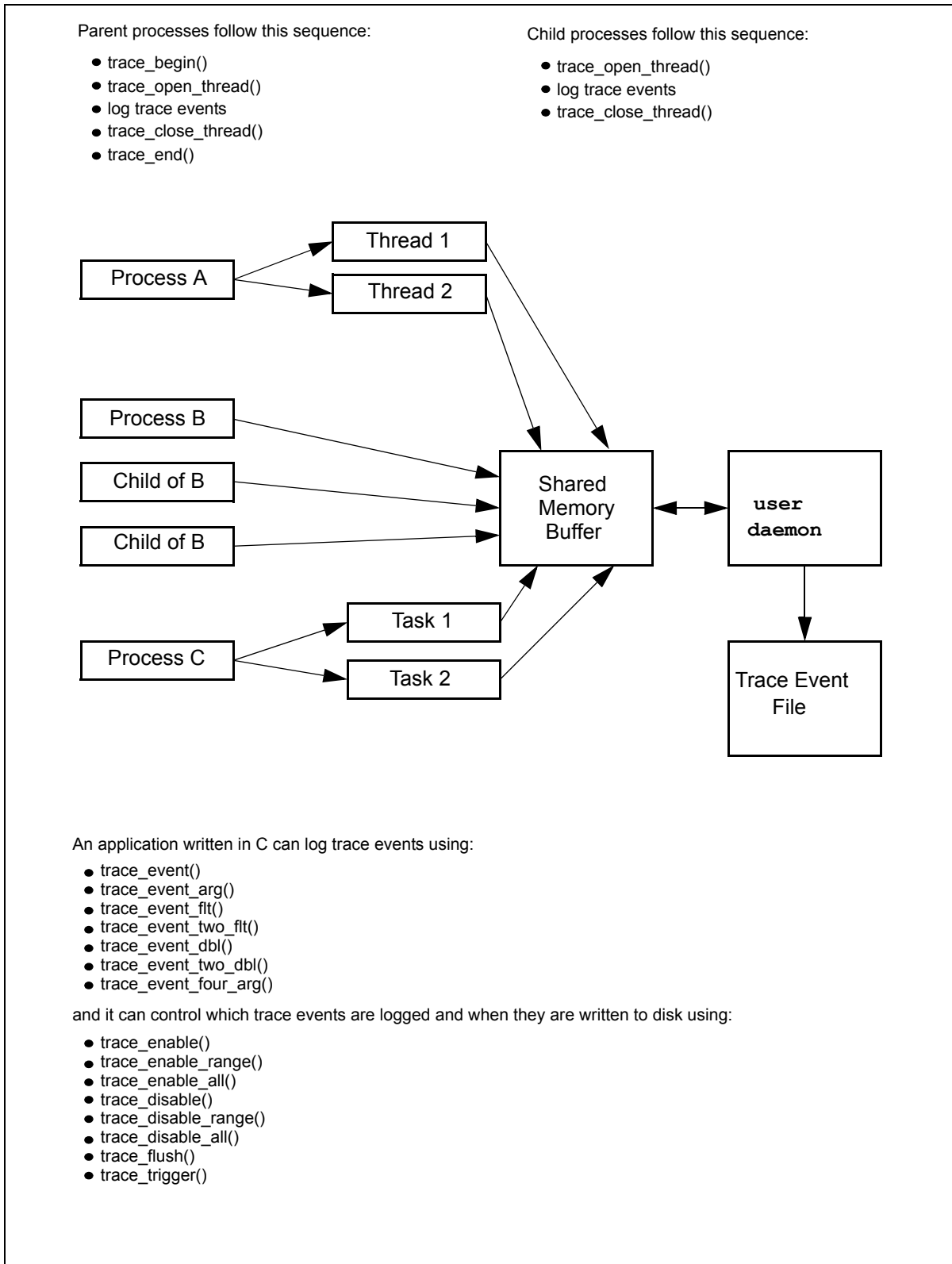


Figure 2-1. Inter-Process Communication and Library Routines



For Fortran, the *config* record should be represented by an array of items. Each member of the array must be provided.

The following describe the individual parameters:

C: *ntc\_buffer\_size*  
Ada: *buffer\_size*  
Fortran: *config(1)*

The size of the shared memory buffer in units of trace events. The user daemon default size is 1024\*16 events. The size must be greater or equal to 4096.

C: *ntc\_use\_spl*  
Ada: *use\_spl*  
Fortran: *config(2)*

Specifies that SPL preemption-control will be used (see **spl\_request(3C)**). This feature is not available on RedHawk Linux systems and is automatically translated to *ntc\_use\_resched*. The user daemon default value is TRUE for PowerMAX OS and FALSE for RedHawk Linux.

C: *ntc\_use\_resched*  
Ada: *use\_resched*  
Fortran: *config(3)*

Specifies that rescheduling variable preemption-control will be used (See **resched\_cntl(2)**). The user daemon default value is FALSE for PowerMAX OS and TRUE for RedHawk Linux.

C: *ntc\_lock\_pages*  
Ada: *lock\_pages*  
Fortran: *config(4)*

Specifies that critical pages will be locked in memory. The user daemon default value is TRUE.

C: *ntc\_clock*  
Ada: *clock*  
Fortran: *config(5)*

Specifies which clock to use as a timing source. This value must be one of NT\_USE\_ARCHITECTURE\_CLOCK or

NT\_USE\_RCIM\_TICK\_CLOCK. The user daemon default value is NT\_USE\_ARCHITECTURE\_CLOCK.

C: *ntc\_shmid\_perm*  
 Ada: *shmid\_perm*  
 Fortran: *config(6)*

Specifies the permissions to use when creating the shared memory segment. The user daemon default value is 0666.

C: *ntc\_daemon\_preferred*  
 Ada: *inherit*  
 Fortran: *config(7)*

Specifies that if a user daemon already exists and the configuration settings differ from these configuration settings, that the user daemon settings are preferred and these values are ignored (although the value of *ntc\_buffer\_size* specified to this routine must not be larger than the size set by the daemon).

## DESCRIPTION

The `trace_begin()` routine performs the following operations:

- Verifies that the version of the NightTrace library linked with the application is compatible with the version used by the user daemon if it is already running
- Verifies the supplied configuration settings are not in conflict with a pre-existing daemon or defines the configuration with these settings if the user daemon does not yet exist.
- Verifies that the RCIM synchronized tick clock is counting if it was selected as the timestamp source
- Attaches the shared memory buffer (after creating it if needed)
- Initialized the preemption control mechanism
- Locks critical NightTrace library routine pages in memory
- Initializes trace event tracing in this process

(PowerMAX Only) For more information on shared memory and the system's interrupt priority level (IPL) register, see the *PowerMAX OS Real-Time Guide*. For information about page-locking privilege (P\_PLOCK), see **intro (2)**.

A process that results from the **exec (2)** system service does not inherit a trace mechanism. Therefore, if that process is to log trace events, it must initialize the trace with `trace_begin()`. Processes that result from a fork in a process that has already initialized the trace need not call `trace_begin()`.

The `trace_begin()` routine must be called only once per parent process (unless a `trace_end()` call has been made).

For processes using C threads and PowerMAX OS Ada tasks, all threads and tasks will inherit the trace context of the `trace_begin()` call that is made by any thread or task of the process.

## RETURN VALUES

Upon successful operation, the `trace_begin()` routine returns `NTNOERROR` or `NTLISTEN`; the latter in the case where no daemon has yet been started. A list of `trace_begin()` return codes follows.

- [`NTNOERROR`] A daemon has already been started that matches the filename passed as *key\_file*. The application can begin to log trace events after calling `trace_open_thread()`.
- [`NTLISTEN`] All operations were successful, but no user daemon matching the filename passed as *key\_file* could be found. The application can continue to make NightTrace API calls but attempts to log events will fail until a daemon is started, at which point logging of events will succeed.
- [`NTALREADY`] The application has already initialized the trace without an intervening `trace_end()`. Tracing can continue in spite of this error. Solution: Remove redundant `trace_begin()` calls.
- [`NTBADVERSION`] The calling application is linked with the static NightTrace library and the static library is not compatible with the NightTrace library being used by the user daemon. Solution: Relink the application with the static library version which matches the library version being used by the daemon.
- [`NTMAPCLOCK`] The selected event timestamp source could not be attached. Solution: If read access is lacking, see your system administrator.  
  
This can also occur if the RCIM synchronized tick clock is selected as the event timestamp source but the tick clock is not counting. Solution: Start the synchronized tick clock by using the `clock_synchronize(1M)` command and restart the application.
- [`NTPERMISSION`] The calling application lacks permission to attach the shared memory buffer. Solution: Make sure that the same user who started the user daemon is the current user logging trace events in the application.
- [`NTMAPSPLREG`] The system's IPL register could not be attached. Solution: If read or write access is lacking, see your system administrator or set `ntc_use_spl` to `FALSE`.
- [`NTPGLOCK`] Permission to lock the text and data pages of the NightTrace library routines was denied. If the user is not privileged to lock pages, see your system administrator or set `ntc_lock_pages` to `FALSE`.

[NTNOSHMID] This can occur if the size of the shared memory buffer exceeds the system limit (`SHMMAX`) or the shared memory buffer already exists but the size required by `ntc_buffer_size` (which is roughly `ntc_buffer_size * sizeof(ntevent_t)`) exceeds the current size.

**SEE ALSO**

Related routines include: `trace_open_thread()`, `trace_end()`

## trace\_open\_thread()

The `trace_open_thread()` routine prepares the current process C thread or Ada task for trace event logging.

### SYNTAX

```
C:          int trace_open_thread(char *thread_name);

Fortran:    integer function trace_open_thread(thread_name)
            character *(*) thread_name

Ada:        function trace_open_thread(
            thread_name : string
            )
            return ntrace_error;
```

### PARAMETERS

*thread\_name*

In NightTrace every thread of execution to be traced (whether a separate process, or a C thread or Ada task within a process) must be associated with a name, *thread\_name*, which may be null. NightTrace's graphical displays and textual summary information show which threads logged trace events. If the `trace_open_thread()` thread name is null, the **ntrace** display utility uses the global thread identifier (TID) as a label in these displays. For more information on global thread identifiers see "Threads" on page 10-52.

Naming your threads can make the displays much more readable. `trace_open_thread()` lets you associate a meaningful character string name with the current threads' more cryptic numeric TID. If you provide a character string as the thread name, the **ntrace** display utility uses it as a label in its displays. Because **ntrace** may be unable to display long strings in the limited screen space available, keep thread names short. (Long thread names cause NightTrace to log an `NT_CONTINUE` overhead trace event.)

The following words are reserved in NightTrace and should not be used in upper case or lower case as thread names: `NONE`, `ALL`, `ALLUSER`, `ALLKERNEL`, `TRUE`, `FALSE`, `CALC`. See "Pre-Defined String s" on page 4-16 for more information about thread names.

### NOTE

Thread names must begin with an alphabetic character and consist solely of alphanumeric characters and the underscore. Spaces and punctuation are not valid characters.



## DESCRIPTION

A NightTrace “thread” can be a process, C thread or Ada task. For **ntrace** displays, `trace_open_thread()` associates a thread name with the process, thread or task logging trace events. Each process, including child processes, that logs trace events must have its own `trace_open_thread()` call. In addition, C threads and Ada tasks may call `trace_open_thread()` individually to associate unique thread names with their trace events. In this way, the different trace contexts of multiple processes, threads and tasks can be easily distinguished from each other.

For more information on threads, see “Programming with the Threads Library” in the *PowerMAX OS Programming Guide*.

A process that results from the **exec (2)** system service does not inherit a trace mechanism. Therefore, if that process is to log trace events, it must call both `trace_begin()` and `trace_open_thread()`.

## RETURN VALUES

The `trace_open_thread()` routine returns a zero value (NTNOERROR) on successful completion. Otherwise, it returns a non-zero value to identify the error condition. A list of `trace_open_thread()` error codes follows.

[NTINIT]	The NightTrace library routines were not initialized or they were initialized but no user daemon has yet been initiated. Ensure a <code>trace_begin()</code> call precedes this call. If the preceding <code>trace_begin()</code> call returned NTLISTEN, then a value of NTINIT is not a failure condition and once a user daemon is started, subsequent attempts at logging events will succeed.
[NTINVALID]	An invalid thread name was specified. Solution: Choose a thread name that meets the requirements mentioned earlier.
[NTRESOURCE]	There are not enough resources to open this thread. Solution: Ask your system administrator to increase the size of the process table.
[NTPGLOCK]	Permission to lock the text and data pages of the NightTrace library routines was denied. If the user has insufficient privileges to lock pages, see the system administrator or specify that page locking is not requested on the <code>trace_begin()</code> call and/or with the user daemon invocation.

## SEE ALSO

Related routines include: `trace_begin()`, `trace_close_thread()`.

## trace\_event() and Its Variants

The following routines log an enabled trace event and possibly some arguments to the shared memory buffer.

## SYNTAX

```
C:      int trace_event (int ID);

        int trace_event_arg (int ID, long arg);

        int trace_event_flt (int ID, float arg);

        int trace_event_two_flt (int ID, float arg1, float arg2);

        int trace_event_dbl (int ID, double arg);

        int trace_event_two_dbl (int ID, double arg1, double arg2);

        int trace_event_four_arg (
        int ID, long arg1, long arg2,
        long arg3, long arg4
        );
```

```
Fortran: integer function trace_event (ID)
         integer ID

         integer function trace_event_arg (ID, arg)
         integer ID, arg

         integer function trace_event_flt (ID, arg)
         integer ID
         real arg

         integer function trace_event_two_flt (ID, arg1, arg2)
         integer ID
         real arg1, arg2

         integer function trace_event_dbl (ID, arg)
         integer ID
         double precision arg

         integer function trace_event_two_dbl (ID, arg1, arg2)
         integer ID
         double precision arg1, arg2

         integer function trace_event_four_arg (ID, arg1, arg2, arg3, arg4)
         integer ID, arg1, arg2, arg3, arg4
```

```
Ada:    type event_type is range 0.4095;
```

(procedures)

```
procedure trace_event (ID : event_type);

procedure trace_event (ID : event_type; arg : integer);

procedure trace_event (ID : event_type; arg : float);
```

```

procedure trace_event (
  ID : event_type;
  arg1 : float; arg2 : float
);

procedure trace_event (ID : event_type; arg : long_float);

procedure trace_event (
  ID : event_type;
  arg1 : long_float; arg2 : long_float
);

procedure trace_event (
  ID : event_type;
  arg1 : integer; arg2 : integer;
  arg3 : integer; arg4 : integer
);

```

**(functions)**

```

function trace_event (ID : event_type)
return ntrace_error;

function trace_event (ID : event_type; arg : integer)
return ntrace_error;

function trace_event (ID : event_type; arg : float)
return ntrace_error;

function trace_event (
  ID : event_type;
  arg1 : float; arg2 : float
)
return ntrace_error;

function trace_event (ID : event_type; arg : long_float)
return ntrace_error;

function trace_event (
  ID : event_type;
  arg1 : long_float; arg2 : long_float
)
return ntrace_error;

```

```
function trace_event (
  ID : event_type;
  arg1 : integer; arg2 : integer;
  arg3 : integer; arg4 : integer
)
return ntrace_error;
```

## PARAMETERS

*ID* Each trace event has a user-defined trace event ID, *ID*. This ID is a valid integer in the range reserved for user trace events (0-4095, inclusive). See “Pre-Defined Strings” on page 4-16 for more information about trace event IDs.

*argN* Sometimes it is useful to log the current value of a variable or expression, *arg*, along with your trace event. The trace event logging routines provide this capability. They differ by how many and what types of numeric arguments they accept. The `trace_event()` routine takes no *args*. The `trace_event_arg()` routine takes a type long *arg*. The `trace_event_flt()` and `trace_event_two_flt` routines take (floating point) type of float *args*. The `trace_event_dbl()` and `trace_event_two_dbl()` routines take (floating point) type double *args*. The `trace_event_four_arg()` routine takes four type long *args*. If you want the **ntrace** display utility to display these trace event arguments in anything but decimal integer format, you can enter the trace event in an event-map file. See “Event Map Files” on page 4-10 for more information on event-map files and formats. Alternatively, you could call the `format()` function. See “format()” on page 11-110 for details.

Every call to `trace_event_four_arg()` causes NightTrace to log an NT\_CONTINUE overhead trace event.

## DESCRIPTION

A *trace point* is a place in your application's source code where you call a trace event logging routine. Usually this location marks a line that is important to debugging or performance analysis. Ideally, trace events provide enough information to be useful, but not so much information that it is overwhelming. Meeting these goals requires careful trace-point planning.

### TIP:

To save time re-editing, recompiling, and relinking your application, consider beginning with a few too many trace points in the source code. You can dynamically enable or disable specific trace events. You can also save time by using **ntrace** options to restrict which trace events are loaded for analysis. See “Command-line Options” on page 4-1 for details.

Some typical trace points include the following:

- Suspected bug locations
- Process, subprogram, or loop entry and exit points
- Timing points, especially for clocking I/O processing
- Synchronization points / multi-process interaction
- Endpoints of atomic operations
- Endpoints of shared memory access code

Call one trace event logging routine at each of the trace points you have selected. When you call this routine, it writes the trace event information (including timings and any arguments) to a shared memory buffer. By default, if this write fills the shared memory buffer or causes the buffer-full cutoff percentage to be reached, the user daemon wakes up and copies the trace event to the trace event file on disk.

Usually each trace event logging routine logs a different trace event ID number. This lets you easily identify which source line logged the trace event, how often that source line executed, and what order source lines executed in. However, it is sometimes useful to log the same trace event ID in multiple places. This makes it possible to group trace events from related, but not identical, activities. In this case, a change of trace event ID usually separates or subdivides groups.

Probably the most common use of trace events is to identify *states*. Two different trace event IDs delimit the boundaries of a state. Most applications log recurring states with different time gaps (from the end of one instance of a state to the start of another) and different state durations (from the start of one instance of a state to its end).

**TIP:**

Consider putting related trace event IDs within a range. Library routines and user daemon options let you manipulate trace events by using trace event ID ranges.

By default, all trace events are enabled for logging. The NightTrace library contains routines that allow you to selectively or globally enable or disable trace events. The user daemon has options that provide similar control. Attempting to log a disabled trace event has no effect. See “`trace_enable()`, `trace_disable()`, and Their Variants” on page 2-17 for more information.

**TIP:**

Consider using symbolic constants instead of numeric trace event IDs. This would make your calls to NightTrace routines more readable.

Once your application logs all of its trace events, you can look at them and their arguments graphically with State Graphs, Event Graphs, and Data Graphs in the `ntrace` display utility. See “State Graph” on page 10-7, “Event Graph” on page 10-6, and “Data Graph” on page 10-8 for more information about these display objects.

## RETURN VALUES

The `trace_event()`, `trace_event_arg()`, `trace_event_dbl()`, and `trace_event_four_arg()` routines return a zero value (NTNOERROR) on successful completion. Otherwise, they return a non-zero value to identify the error condition. A list of error codes for these routines follows.

- |              |   |
|--------------|---|
| [NTINVALID]  | An invalid trace event ID has been supplied. Solution: Use trace event IDs only in the range 0-4095, inclusive.   |
| [NTINIT]     | The NightTrace library routines were not initialized or they were initialized but no user daemon has yet been initiated. Ensure a <code>trace_begin()</code> and <code>trace_open_thread()</code> call precede the trace event logging routine call. Once a user daemon is started, subsequent attempts at logging events will succeed. |
| [NTLOSTDATA] | The trace event was lost because the shared memory buffer was full. This can occur if the user daemon cannot empty the shared memory buffer quickly enough. Increase the priority of the user daemon and/or schedule it on a CPU with less activity. Additionally, the size of the shared memory buffer can be increased.               |

## SEE ALSO

Related routines include:

```
trace_flush(), trace_trigger(),  
trace_enable(), trace_enable_range(),  
trace_enable_all(), trace_disable(),  
trace_disable_range(), trace_disable_all()
```

## trace\_enable(), trace\_disable(), and Their Variants

By default, all trace events are enabled for logging to the shared memory buffer. The `trace_disable()`, `trace_disable_range()`, and `trace_disable_all()` routines respectively make your application ignore requests to log one or more trace events. The `trace_enable()`, `trace_enable_range()`, and `trace_enable_all()` routines respectively make your application notice previously disabled requests to log one or more trace events.

### SYNTAX

```
C:      int trace_enable (int ID);

        int trace_enable_range (int ID_low, int ID_high);

        int trace_enable_all ();

        int trace_disable (int ID);

        int trace_disable_range (int ID_low, int ID_high);

        int trace_disable_all ();
```

```
Fortran: integer function trace_enable (ID)
          integer ID

          integer function trace_enable_range (ID_low, ID_high)
          integer ID_low, ID_high

          integer function trace_enable_all ()

          integer function trace_disable (ID)
          integer ID

          integer function trace_disable_range (ID_low, ID_high)
          integer ID_low, ID_high

          integer function trace_disable_all ()
```

```
Ada:    type event_type is range 0..4095;
```

(procedures)

```
procedure trace_enable (ID : event_type);

procedure trace_enable (
  ID_low : event_type; ID_high : event_type
);

procedure trace_enable_all;

procedure trace_disable (ID : event_type);
```

```
procedure trace_disable (  
  ID_low : event_type; ID_high : event_type  
);  
  
procedure trace_disable_all;
```

(functions)

```
function trace_enable (ID : event_type)  
return ntrace_error;  
  
function trace_enable (  
  ID_low : event_type; ID_high : event_type  
)  
return ntrace_error;  
  
function trace_enable_all  
return ntrace_error;  
  
function trace_disable (ID : event_type)  
return ntrace_error;  
  
function trace_disable (  
  ID_low : event_type; ID_high : event_type  
)  
return ntrace_error;  
  
function trace_disable_all  
return ntrace_error;
```

## PARAMETERS

- |                |  |
|----------------|--|
| <i>ID</i>      | Each trace event has a user-defined trace event ID, <i>ID</i> . This ID is a valid integer in the range reserved for user trace event IDs (0-4095, inclusive). See “trace_event() and Its Variants” on page 2-11 for more information. |
| <i>ID_low</i>  | It is possible to manipulate groups of trace event IDs by specifying a range of trace event IDs. <i>ID_low</i> is the smallest trace event ID in the range.  |
| <i>ID_high</i> | It is possible to manipulate groups of trace event IDs by specifying a range of trace event IDs. <i>ID_high</i> is the largest trace event ID in the range.  |

## DESCRIPTION

The enable and disable library routines allow you to select which trace events are enabled and which are disabled for logging. A discussion of disabling trace events appears first because initially all trace events are enabled.

Sometimes, so many trace events that it is hard to understand the **ntrace** display. Occasionally you know that a particular trace event or trace event range is not interesting at certain times but is interesting at others. When either of these conditions exist, it is useful to disable the extraneous trace events. You can disable trace events



temporarily, where you disable and later re-enable them. You can also disable them permanently, where you disable them at the beginning of the process or at a later point and never re-enable them.

## NOTE

These routines enable and disable trace events in all processes that rely on the same user daemon to log to the same trace event file.

All disable library routines make your application start ignoring requests to log trace event(s) to the shared memory buffer. The disable routines differ by how many trace events they disable. `trace_disable()` disables one trace event ID. `trace_disable_range()` disables a range of trace event IDs, including both range endpoints. `trace_disable_all()` disables all trace events. Disabling an already disabled trace event has no effect.

All enable library routines let you re-enable a trace event that you disabled with a disable library routine or user daemon. The effect is that your application resumes noticing requests to log the specified trace event to the shared memory buffer. The enable routines differ by how many trace events they enable. `trace_enable()` enables one trace event ID. `trace_enable_range()` enables a range of trace event IDs, including both range endpoints. `trace_enable_all()` enables all trace events. Enabling an already enabled trace event has no effect.

### TIP:

Consider invoking the user daemon with events disabled instead of calling the `trace_enable()` and `trace_disable()` routines. Using these options saves you from re-editing, recompiling and relinking your application.

### TIP:

If you want to log only a few of your trace events, disable all trace events with `trace_disable_all()` and then selectively enable the trace events of interest.

## RETURN VALUES

The `trace_disable()`, `trace_disable_range()`, `trace_disable_all()`, `trace_enable()`, `trace_enable_range()`, and `trace_enable_all()` routines return a zero value (NTNOERROR) on successful completion. Otherwise, they return a non-zero value to identify the error condition. A list of error codes for these routines follows.

- |             |   |
|-------------|---|
| [NTINIT]    | The NightTrace library routines were not initialized. Solution: Be sure a <code>trace_begin()</code> and <code>trace_open_thread()</code> call precede the call to the disable or enable routine. |
| [NTINVALID] | An invalid trace event ID has been supplied. Solution: Use trace event IDs only in the range 0-4095, inclusive.   |

**SEE ALSO**

Related routines include:

```
trace_event(), trace_event_arg(),  
trace_event_dbl(), trace_event_four_arg()
```

## trace\_flush() and trace\_trigger()

The `trace_flush()` and `trace_trigger()` routines asynchronously wake the user and direct it to copy trace events from the shared memory buffer to the trace event file on disk. Note: These routines do not wait for the copy to complete.

### SYNTAX

C:

```
int trace_flush();
int trace_trigger();
```

Fortran:

```
integer function trace_flush()
integer function trace_trigger()
```

Ada:  
(procedures)

```
procedure trace_flush;
procedure trace_trigger;
```

(functions)

```
function trace_flush
return ntrace_error;

function trace_trigger
return ntrace_error;
```

### DESCRIPTION

When the user daemon is idle, it sleeps. The process of copying trace events from the shared memory buffer to a trace event file is called *flushing the buffer*. The user daemon wakes up and flushes the buffer when any of these conditions exist:

- the user daemon's sleep interval elapses
- The buffer-full cutoff percentage is exceeded
- The shared memory buffer is full of unwritten trace events
- Your application calls `trace_flush()`, `trace_trigger()`, or `trace_end()`
- No event has been logged in a period of time in which the lower 32 bits of the timestamp source would roll over. It is important to detect this rollover so that proper ordering of trace events is maintained.

User daemon options let you set limits for the first three conditions above. When you invoke a user daemon with one of these options and it detects the corresponding condition, it automatically flushes the buffer. There is one key way that `trace_flush()` and `trace_trigger()` differ from the flush control the user daemon provides: with `trace_flush()` and `trace_trigger()` you decide when to asynchronously flush the shared memory buffer based on your program flow, and with certain options the user daemon flushes the shared memory buffer automatically.

If the shared memory buffer becomes full of trace events, trace events may be lost. To keep this situation from occurring, configure the user daemon to flush the buffer regularly. This is particularly good to do if your application will soon be busy.

Waking the user daemon to flush the buffer takes time and this overhead can distort trace event timings. Therefore, call `trace_flush()` and `trace_trigger()` only in parts of your application where time is not critical.

**TIP:**

`trace_trigger()` is identical to `trace_flush()`, except `trace_trigger()` works only in buffer-wraparound mode. Call `trace_trigger()` instead of `trace_flush()` so that only buffer-wraparound's performance is affected.

When you run in buffer-wraparound mode, you are telling NightTrace to intentionally discard older or less-vital trace events when the shared memory buffer gets full. In buffer-wraparound mode, you must explicitly call `trace_flush()` or `trace_trigger()`. Only then, does the user daemon copy the remaining trace events from the shared memory buffer to the trace event file. However, do not call `trace_flush()` or `trace_trigger()` too often or you will reduce the effectiveness of this mode. See "Option to Establish Buffer-Wraparound Mode (-buffer-wrap)" on page 6-13 for more information on buffer-wraparound mode.

**RETURN VALUES**

The `trace_flush()` and `trace_trigger()` routines return a zero value (NTNOERROR) on successful completion. Otherwise, they return a non-zero value to identify the error condition. A list of `trace_flush()` and `trace_trigger()` error codes follows.

[NTFLUSH]            A failure occurred while attempting to flush the shared memory buffer. Solution: Verify the status of the user daemon; if necessary, restart it and rerun the trace.

**SEE ALSO**

Related routines include:

`trace_event()`, `trace_event_arg()`,  
`trace_event_dbl()`, `trace_event_four_arg()`

## trace\_close\_thread()

The `trace_close_thread()` routine disables trace event logging for the current thread or process.

### SYNTAX

```
C:          int trace_close_thread();

Fortran:    integer function trace_close_thread()

Ada:        function trace_close_thread return
            ntrace_error;
```

### DESCRIPTION

A NightTrace *thread* can be a process, C thread or Ada task. Each thread that C calls `trace_open_thread()` must have its own `trace_close_thread()` call. For more information on threads, see “Programming with the Threads Library” in the *PowerMAX OS Programming Guide*.

### RETURN VALUES

The `trace_close_thread()` routine returns a zero value (NTNOERROR) on successful completion. Otherwise, it returns a non-zero value to identify the error condition. A list of `trace_close_thread()` error codes follows.

[NTINIT]	The NightTrace library routines were not initialized. Solution: Call <code>trace_close_thread()</code> only once if you previously called <code>trace_open_thread()</code> .
----------	--

### SEE ALSO

Related routines include: `trace_open_thread()`, `trace_end()`

## trace\_end()

The `trace_end()` routine frees resources and terminates trace event tracing in your process.

### SYNTAX

```
C:          int trace_end();

Fortran:    integer function trace_end()

Ada:        function trace_end
            return ntrace_error;
```

### DESCRIPTION

Generally, call `trace_end()` only once per logging process. However, for processes using C threads or Ada tasks, `trace_end()` must also be called by any individual threads or tasks that have previously called `trace_begin()`. `trace_end()` performs the following operations:

- Terminates trace event tracing in this process or thread
- Flushes trace events from the shared memory buffer to the trace event file
- Detaches the shared memory buffer, timestamp source, and interrupt priority level (IPL) register
- Notifies the user daemon that the current process has finished logging trace events

### RETURN VALUES

The `trace_end()` routine returns a zero value (`NTNOERROR`) on successful completion. Otherwise, it returns a non-zero value to identify the error condition. A list of `trace_end()` error codes follows.

[NTFLUSH]            A failure occurred while attempting to flush the shared memory buffer. Solution: Verify the status of the user daemon; if necessary, restart it and rerun the trace.

[NTNODAEMON]        There is no user daemon with a trace event file name that matches the one on the `trace_begin()` call attached to the shared memory region. This condition is not always detectable. Solution: Use the `ntrace` display utility to analyze your logged trace events.

### SEE ALSO

Related routines include: `trace_begin()`, `trace_close_thread()`

## Disabling Tracing

There are four ways to disable tracing in your application:

- For C applications, put a `#include <ntrace.h>` in your source code. You must either recompile your application with the `-DNTRACE` preprocessor option or insert the following preprocessor control statement before the `#include <ntrace.h>`.

```
#define NNTRACE
```

The NightTrace header file, `ntrace.h`, contains macro counterparts for each NightTrace library routine. When you define `NNTRACE`, the compiler treats your NightTrace routine calls as if they were macro calls that always return a success (zero) status. For more information on preprocessor options, see `cpp(1)`.

Use a command similar to the following one to turn off tracing in your application, `fl_sim.c`.

```
$ cc -DNTRACE fl_sim.c -lud
```

By disabling tracing this way, you have to rebuild your application, but you save compilation and execution time.

- Call the `trace_disable_all()` routine near the top of the source, recompile, and relink your application with the NightTrace library. (For more information about this routine, see “`trace_enable()`, `trace_disable()`, and Their Variants” on page 2-17.) If your application calls any of the enable routines, this method is not entirely effective.

By disabling tracing this way, you have to rebuild your application, and there is no saving in compilation time or execution time.

- Start a user daemon with all events disabled.

By disabling tracing this way, you do not have to rebuild your application, but there is no saving in compilation time or execution time.

- Do not start a user daemon.

By disabling tracing this way, you do not have to rebuild your application, but there is no saving in compilation or execution time.

## Compiling and Linking

You must link in the NightTrace library so that your application can initialize its trace mechanism and log trace events. The name of this library depends on your source language. C and Fortran applications must link in the `/usr/lib/libntrace.a` library.

## C Example

PowerMAX OS:

```
$ cc fl_sim.c -lntrace -lud
```

RedHawk Linux:

```
$ cc fl_sim.c -lntrace -lccur_rt
```

This step:

- Compiles the `fl_sim.c` application
- Links in the NightTrace library
- Creates an executable named `a.out` if there were no major errors

## Fortran Example

PowerMAX OS:

```
$ hf77 turn_matrix.f -lntrace -lud
```

RedHawk Linux:

```
$ cf77 turn_matrix.f -lntrace -lccur_rt
```

This step:

- Compiles the `turn_matrix.f` application
- Links in the NightTrace library
- Creates an executable named `a.out` if there were no major errors

For more information on compiling and linking Concurrent Fortran 77 programs, see the *Concurrent Fortran 77 Reference Manual* (0890240).

## Ada Example

For a complete example on accessing the NightTrace library routines from an Ada application, see the section titled “NightTrace Binding” in the *MAXAda Reference Manual*.



## Using the NightTrace Analysis API

NightTrace Analysis Application Programming Interface . . . . .	3-1
Data Structures . . . . .	3-3
tr_cb_t . . . . .	3-3
tr_cond_cb_func_t . . . . .	3-4
tr_cond_func_t . . . . .	3-4
tr_cond_t . . . . .	3-5
tr_dir_t . . . . .	3-5
tr_offset_t . . . . .	3-5
tr_state_action_t . . . . .	3-6
tr_state_cb_func_t . . . . .	3-6
tr_state_info_t . . . . .	3-7
tr_state_t . . . . .	3-7
tr_stream_event_t . . . . .	3-8
tr_stream_func_t . . . . .	3-8
tr_string_node_t . . . . .	3-8
tr_t . . . . .	3-8
Functions . . . . .	3-9
API Initialization and Destruction . . . . .	3-13
tr_init() . . . . .	3-13
tr_destroy() . . . . .	3-13
Error Detection, Collection, and Reporting . . . . .	3-15
tr_error_clear() . . . . .	3-15
tr_error_check() . . . . .	3-16
Input Specification and Streaming Control . . . . .	3-17
tr_open_file() . . . . .	3-17
tr_open_stream() . . . . .	3-18
tr_close() . . . . .	3-19
tr_stream_notify() . . . . .	3-20
tr_stream_consume() . . . . .	3-21
tr_stream_size() . . . . .	3-21
Event Offset Positioning . . . . .	3-23
tr_next_event() . . . . .	3-23
tr_next_event_() . . . . .	3-24
tr_prev_event() . . . . .	3-24
tr_prev_event_() . . . . .	3-25
tr_search() . . . . .	3-26
tr_seek() . . . . .	3-27
Basic Event Attribute Functions . . . . .	3-28
tr_id() . . . . .	3-29
tr_id_() . . . . .	3-29
tr_time() . . . . .	3-30
tr_time_() . . . . .	3-31
tr_nargs() . . . . .	3-31
tr_nargs_() . . . . .	3-32
tr_arg_int() . . . . .	3-33
tr_arg_int_() . . . . .	3-33
tr_arg_dbl() . . . . .	3-34

tr_arg_dbl_()	3-35
tr_pid()	3-35
tr_pid_()	3-36
tr_raw_pid()	3-38
tr_raw_pid_()	3-39
tr_lwpid()	3-40
tr_lwpid_()	3-41
tr_tid()	3-41
tr_tid_()	3-42
tr_thread_id()	3-43
tr_thread_id_()	3-43
tr_task_id()	3-44
tr_task_id_()	3-45
tr_cpu()	3-45
tr_cpu_()	3-46
tr_node()	3-47
tr_node_()	3-48
tr_process_name()	3-48
tr_process_name_()	3-49
tr_task_name()	3-50
tr_task_name_()	3-50
tr_thread_name()	3-51
tr_thread_name_()	3-51
Conditions	3-53
tr_cond_create()	3-54
tr_cond_reset()	3-55
tr_cond_find()	3-55
tr_cond_id()	3-56
tr_cond_id_range()	3-57
tr_cond_id_clear()	3-58
tr_cond_cpu()	3-59
tr_cond_cpu_clear()	3-59
tr_cond_pid()	3-60
tr_cond_pid_name()	3-61
tr_cond_pid_clear()	3-62
tr_cond_tid()	3-63
tr_cond_tid_name()	3-64
tr_cond_tid_clear()	3-65
tr_cond_node()	3-66
tr_cond_node_clear()	3-67
tr_cond_func_or()	3-68
tr_cond_func_and()	3-70
tr_cond_func_clear()	3-72
tr_cond_expr_and()	3-73
tr_cond_expr_or()	3-74
tr_cond_not()	3-75
tr_cond_or()	3-76
tr_cond_and()	3-77
tr_cond_copy()	3-78
tr_cond_name()	3-79
tr_cond_satisfy()	3-79
tr_cond_satisfy_()	3-80
tr_cond_register()	3-81
tr_cond_offset()	3-82

State-oriented Interfaces . . . . .	3-83
tr_state_create() . . . . .	3-84
tr_state_find() . . . . .	3-85
tr_state_name() . . . . .	3-85
tr_state_start_id() . . . . .	3-86
tr_state_start_id_range() . . . . .	3-87
tr_state_start_id_clear() . . . . .	3-88
tr_state_end_id() . . . . .	3-88
tr_state_end_id_range() . . . . .	3-89
tr_state_end_id_clear() . . . . .	3-90
tr_state_start_cond() . . . . .	3-90
tr_state_start_cond_clear() . . . . .	3-91
tr_state_end_cond() . . . . .	3-92
tr_state_end_cond_clear() . . . . .	3-92
tr_activate() . . . . .	3-93
tr_state_info() . . . . .	3-94
tr_state_info_() . . . . .	3-95
tr_state_active() . . . . .	3-96
tr_state_active_() . . . . .	3-97
Output Function . . . . .	3-98
tr_copy_input() . . . . .	3-98
String Table Functions . . . . .	3-99
tr_get_string() . . . . .	3-99
tr_get_item() . . . . .	3-100
tr_create_table() . . . . .	3-101
tr_append_table() . . . . .	3-102
Callback Interfaces . . . . .	3-103
tr_iterate() . . . . .	3-103
tr_halt() . . . . .	3-104
tr_cancel_cb() . . . . .	3-104
tr_cond_cb() . . . . .	3-105
tr_state_cb() . . . . .	3-106
Sample Programs . . . . .	3-107
list . . . . .	3-108
list.c . . . . .	3-108
search . . . . .	3-110
search.c . . . . .	3-110
watchdog . . . . .	3-112
watchdog.c . . . . .	3-112
ptime . . . . .	3-115
ptime.c . . . . .	3-116
browse . . . . .	3-118
browse.c . . . . .	3-118



## Using the NightTrace Analysis API

---

The NightTrace graphical user interface is one of the primary tools for analyzing trace data (see Chapter 5, “Using the NightTrace GUI”). However, the NightTrace Analysis Application Programming Interface provides users with even further control in summarizing or monitoring trace data.

The NightTrace Analysis API provides a basic interface to the data produced by NightTrace allowing users to process NightTrace data programmatically. It allows users to customize their analysis of NightTrace data, both expressly via user-written programs and as customized batch summaries.

For instance, a user may want to provide customized reports on user application or kernel activity, monitor a user application or the operating system itself and take action when a specific situation occurs, or filter a trace data file (to significantly reduce its size) for subsequent use with the GUI or API.

The NightTrace Analysis API can use either NightTrace data files generated by NightTrace kernel or user daemons or may reference a file descriptor connected to a streaming daemon as the input source.

The API allows the user to control the order in which the data is accessed and provides for event filtration as well as customized event and state definition specification using conditions currently provided in the NightTrace GUI tool.

In addition, all functions supported by the NightTrace GUI expression language are provided as user-callable functions.

The following sections describe the data structures and functions that comprise the NightTrace Analysis API.

Sample programs using these data structures and functions are also provided (see “Sample Programs” on page 3-107).

## NightTrace Analysis Application Programming Interface

The NightTrace Analysis Application Programming Interface consists of a number of data structures (see “Data Structures” on page 3-3) and functions (see “Functions” on page 3-9).

These data structures and functions are accessible via the C header file:

```
/usr/include/ntrace_analysis.h
```

and the C library:

```
/usr/lib/libntrace_analysis.a
```

and can be called by C and C++ programs.

## Data Structures

The following data structures are part of the NightTrace Analysis Application Programming Interface:

- `tr_cb_t` (see page 3-3)
- `tr_cond_cb_func_t` (see page 3-4)
- `tr_cond_func_t` (see page 3-4)
- `tr_cond_t` (see page 3-5)
- `tr_dir_t` (see page 3-5)
- `tr_offset_t` (see page 3-5)
- `tr_state_action_t` (see page 3-6)
- `tr_state_info_t` (see page 3-7)
- `tr_state_t` (see page 3-7)
- `tr_stream_event_t` (see page 3-8)
- `tr_string_node_t` (see page 3-8)
- `tr_t` (see page 3-8)

See “Functions” on page 3-9 for information about the functions available in the NightTrace Analysis API.

### **tr\_cb\_t**

`tr_cb_t` is an opaque handle that identifies a particular callback. It is defined as:

```
typedef int tr_cb_t;
```

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

## tr\_cond\_cb\_func\_t

tr\_cond\_cb\_func\_t is defined as:

```
typedef void (*tr_cond_cb_func_t) (tr_t      t,  
                                   tr_cond_t  c,  
                                   tr_offset_t offset,  
                                   int         occurrence,  
                                   void        * context,  
                                   int         * disable);
```

### PARAMETERS

<i>t</i>	data set handle
<i>c</i>	handle of the condition associated with this call
<i>offset</i>	offset of the trace event satisfying the condition
<i>occurrence</i>	number of times the condition has been satisfied thus far
<i>context</i>	user-defined field specified when the callback is defined
<i>disable</i>	pointer to an integer; if the user sets the integer to a non-zero value, the registration of this function for the specified condition will be disabled for the remainder of the iteration pass

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_offset\_t” on page 3-5

## tr\_cond\_func\_t

tr\_cond\_func\_t is defined as:

```
typedef int (*tr_cond_func_t) (tr_t t,  
                               tr_offset_t event_offset,  
                               void *context);
```

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.



## **tr\_cond\_t**

`tr_cond_t` is an opaque handle used to identify a particular condition. It is defined as:

```
typedef long tr_cond_t;
```

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

## **tr\_dir\_t**

`tr_dir_t` is defined as:

```
typedef enum {tr_forward, tr_backward} tr_dir_t;
```

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

## **tr\_offset\_t**

`tr_offset_t` is defined as:

```
typedef int tr_offset_t;
```

Values of type `tr_offset_t` represent the offset (aka position) of a trace event within the data set. Event offsets are assigned as monotonically increasing integers, starting with zero as the offset of the first event in the data set.

Functions which return `tr_offset_t` may return `TR_EOF`, which indicates exceeding past either the beginning or end of the data set, respectively.

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

**tr\_state\_action\_t**

`tr_state_action_t` is an enumerated type which is used to specify when a certain function will be called. It is defined as:

```
typedef enum { tr_state_start_action,
              tr_state_end_action,
              tr_state_active_action,
              tr_state_inactive_action }
              tr_state_action_t;
```

where:

`tr_state_start_action`

called for every event which starts the state

`tr_state_end_action`

called for every event which ends an active state

`tr_state_active_action`

called for every event for which the state is active

`tr_state_inactive_action`

called for every event for which the state is inactive

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

**tr\_state\_cb\_func\_t**

`tr_state_cb_func_t` is defined as:

```
typedef void (*tr_state_cb_func_t) (tr_t      t,
                                   tr_state_t state,
                                   tr_offset_t offset,
                                   int        occurrence,
                                   void      * context,
                                   int        * disable);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>state</i>	handle of the state associated with this call
<i>offset</i>	offset of the trace event satisfying the condition
<i>occurrence</i>	number of times the condition has been satisfied thus far

<i>context</i>	user-defined field specified when the callback is defined
<i>disable</i>	pointer to an integer; if the user sets the integer to a non-zero value, the registration of this function for the specified state will be disabled for the remainder of the iteration pass

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

## **tr\_state\_info\_t**

`tr_state_info_t` is defined as:

```
typedef struct {
    tr_offset_t start_offset;
    tr_offset_t end_offset;
    double gap;
    double duration;
    int count;
} tr_state_info_t;
```

where:

<code>start_offset</code>	offset of the event that started the specified state
<code>end_offset</code>	offset of the event that ended the specified state
<code>gap</code>	time in seconds between the beginning of the last instance of the specified state and the end of the previous instance (or zero if no previous instance exists)
<code>duration</code>	time in seconds during which the specified state was active
<code>count</code>	number of completed instances of the specified state

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

## **tr\_state\_t**

`tr_state_t` is an opaque handle used to identify a particular state. It is defined as:

```
typedef long tr_state_t;
```

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

## **tr\_stream\_event\_t**

tr\_stream\_event\_t is defined as:

```
typedef enum { tr_stream_overflow,  
              tr_stream_stall } tr_stream_event_t;
```

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

## **tr\_stream\_func\_t**

tr\_stream\_func\_t is defined as:

```
typedef void (*tr_stream_func_t) (tr_t t,  
                                  tr_stream_event_t event);
```

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

## **tr\_string\_node\_t**

tr\_string\_node\_t is defined as:

```
typedef struct {  
    int item;  
    char * value;  
} tr_string_node_t;
```

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

## **tr\_t**

tr\_t is an opaque handle used to identify a particular data set. It is defined as:

```
typedef long tr_t;
```

See “Data Structures” on page 3-3 for other data structures included in the NightTrace Analysis API.

## Functions

The functions that comprise the NightTrace Analysis Application Programming Interface are broken down into the following categories:

- API Initialization and Destruction (see page 3-13)
- Error Detection, Collection, and Reporting (see page 3-15)
- Input Specification and Streaming Control (see page 3-17)
- Event Offset Positioning (see page 3-23)
- Basic Event Attribute Functions (see page 3-28)
- Conditions (see page 3-53)
- State-oriented Interfaces (see page 3-83)
- Output Function (see page 3-98)
- String Table Functions (see page 3-99)
- Callback Interfaces (see page 3-103)

The following is a complete list of functions included in the NightTrace Analysis API:

<code>tr_activate()</code>	page 3-93
<code>tr_append_table()</code>	page 3-102
<code>tr_arg_dbl()</code>	page 3-34
<code>tr_arg_dbl_()</code>	page 3-35
<code>tr_arg_int()</code>	page 3-33
<code>tr_arg_int_()</code>	page 3-33
<code>tr_cancel_cb()</code>	page 3-104
<code>tr_close()</code>	page 3-19
<code>tr_cond_and()</code>	page 3-77
<code>tr_cond_cb()</code>	page 3-105
<code>tr_cond_copy()</code>	page 3-78
<code>tr_cond_cpu()</code>	page 3-59
<code>tr_cond_cpu_clear()</code>	page 3-59
<code>tr_cond_create()</code>	page 3-54
<code>tr_cond_expr_and()</code>	page 3-73
<code>tr_cond_expr_or()</code>	page 3-74
<code>tr_cond_find()</code>	page 3-55
<code>tr_cond_func_and()</code>	page 3-70

<code>tr_cond_func_clear()</code>	page 3-72
<code>tr_cond_func_or()</code>	page 3-68
<code>tr_cond_id()</code>	page 3-56
<code>tr_cond_id_clear()</code>	page 3-58
<code>tr_cond_id_range()</code>	page 3-57
<code>tr_cond_name()</code>	page 3-79
<code>tr_cond_node()</code>	page 3-66
<code>tr_cond_node_clear()</code>	page 3-67
<code>tr_cond_not()</code>	page 3-75
<code>tr_cond_offset()</code>	page 3-82
<code>tr_cond_or()</code>	page 3-76
<code>tr_cond_pid()</code>	page 3-60
<code>tr_cond_pid_clear()</code>	page 3-62
<code>tr_cond_pid_name()</code>	page 3-61
<code>tr_cond_register()</code>	page 3-81
<code>tr_cond_reset()</code>	page 3-55
<code>tr_cond_satisfy()</code>	page 3-79
<code>tr_cond_satisfy_()</code>	page 3-80
<code>tr_cond_tid()</code>	page 3-63
<code>tr_cond_tid_clear()</code>	page 3-65
<code>tr_cond_tid_name()</code>	page 3-64
<code>tr_copy_input()</code>	page 3-98
<code>tr_cpu()</code>	page 3-45
<code>tr_cpu_()</code>	page 3-46
<code>tr_create_table()</code>	page 3-101
<code>tr_destroy()</code>	page 3-13
<code>tr_error_check()</code>	page 3-16
<code>tr_error_clear()</code>	page 3-15
<code>tr_get_item()</code>	page 3-100
<code>tr_get_string()</code>	page 3-99
<code>tr_halt()</code>	page 3-104
<code>tr_id()</code>	page 3-29
<code>tr_id_()</code>	page 3-29
<code>tr_init()</code>	page 3-13
<code>tr_iterate()</code>	page 3-103
<code>tr_lwpid()</code>	page 3-40

<code>tr_lwpid_()</code>	page 3-41
<code>tr_nargs()</code>	page 3-31
<code>tr_nargs_()</code>	page 3-32
<code>tr_next_event()</code>	page 3-23
<code>tr_next_event_()</code>	page 3-24
<code>tr_node()</code>	page 3-47
<code>tr_node_()</code>	page 3-48
<code>tr_open_file()</code>	page 3-17
<code>tr_open_stream()</code>	page 3-18
<code>tr_pid()</code>	page 3-35
<code>tr_pid_()</code>	page 3-36
<code>tr_prev_event()</code>	page 3-24
<code>tr_prev_event_()</code>	page 3-25
<code>tr_process_name()</code>	page 3-48
<code>tr_process_name_()</code>	page 3-49
<code>tr_raw_pid()</code>	page 3-38
<code>tr_raw_pid_()</code>	page 3-39
<code>tr_search()</code>	page 3-26
<code>tr_seek()</code>	page 3-27
<code>tr_state_active()</code>	page 3-96
<code>tr_state_active_()</code>	page 3-97
<code>tr_state_cb()</code>	page 3-106
<code>tr_state_create()</code>	page 3-84
<code>tr_state_end_cond()</code>	page 3-92
<code>tr_state_end_cond_clear()</code>	page 3-92
<code>tr_state_end_id()</code>	page 3-88
<code>tr_state_end_id_clear()</code>	page 3-90
<code>tr_state_end_id_range()</code>	page 3-89
<code>tr_state_find()</code>	page 3-85
<code>tr_state_info()</code>	page 3-94
<code>tr_state_info_()</code>	page 3-95
<code>tr_state_name()</code>	page 3-85
<code>tr_state_start_cond()</code>	page 3-90
<code>tr_state_start_cond_clear()</code>	page 3-91
<code>tr_state_start_id()</code>	page 3-86
<code>tr_state_start_id_clear()</code>	page 3-88

<code>tr_state_start_id_range()</code>	page 3-87
<code>tr_stream_consume()</code>	page 3-21
<code>tr_stream_notify()</code>	page 3-20
<code>tr_stream_size()</code>	page 3-21
<code>tr_task_id()</code>	page 3-44
<code>tr_task_id_()</code>	page 3-45
<code>tr_task_name()</code>	page 3-50
<code>tr_task_name_()</code>	page 3-50
<code>tr_thread_id()</code>	page 3-43
<code>tr_thread_id_()</code>	page 3-43
<code>tr_thread_name()</code>	page 3-51
<code>tr_thread_name_()</code>	page 3-51
<code>tr_tid()</code>	page 3-41
<code>tr_tid_()</code>	page 3-42
<code>tr_time()</code>	page 3-30
<code>tr_time_()</code>	page 3-31



## API Initialization and Destruction

The functions related to API initialization and destruction are:

- `tr_init()` (see page 3-13)
- `tr_destroy()` (see page 3-13)

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### **tr\_init()**

`tr_init()` returns an opaque handle that is required for all subsequent API functions and which identifies the data set.

#### **SYNTAX**

```
extern tr_t tr_init (void);
```

#### **RETURN VALUES**

Returns an opaque handle that is required for all subsequent API functions and which identifies the data set; in the event there is insufficient memory, `TR_NO_HANDLE` will be returned.

See “API Initialization and Destruction” on page 3-13 for related functions. See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### **SEE ALSO**

- “tr\_t” on page 3-8

### **tr\_destroy()**

`tr_destroy()` frees up any remaining memory associated with a handle returned by `tr_init()`.

## NOTE

`tr_destroy()` expects a pointer to a handle, whereas all other functions expect the handle itself.

## SYNTAX

```
extern void tr_destroy (tr_t * t);
```

## PARAMETERS

*t*                      data set handle

See “API Initialization and Destruction” on page 3-13 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8
- “tr\_init()” on page 3-13

## Error Detection, Collection, and Reporting

Most individual functions within the API return an indication of whether the requested operation was successful. Most often, zero indicates success, and non-zero indicates failure. Exceptions to this rule are indicated for each function.

Errors are collected by the API and can be retrieved after calling a series of functions.

The functions related to error detection, collection, and reporting are:

- `tr_error_clear()` (see page 3-15)
- `tr_error_check()` (see page 3-16)

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### `tr_error_clear()`

`tr_error_clear()` is used to flush any collected errors and set the internal error state to zero, meaning success.

#### SYNTAX

```
extern void tr_error_clear (tr_t t);
```

#### PARAMETERS

*t*                      data set handle

See “Error Detection, Collection, and Reporting” on page 3-15 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### SEE ALSO

- “`tr_t`” on page 3-8
- “`tr_error_check()`” on page 3-16

## tr\_error\_check()

`tr_error_check()` is used to determine the errors that have occurred since the beginning of the program or since the last time the error list was cleared.

### SYNTAX

```
extern int tr_error_check (tr_t t,  
                          tr_string_node_t**list);
```

### PARAMETERS

<i>t</i>	data set handle
<i>list</i>	the list of errors that have occurred (since the last call to <code>tr_error_clear()</code> or the beginning of the program). For each entry in the <i>list</i> , <i>value</i> describes the error and <i>item</i> refers to <code>errno</code> (if appropriate). (See “ <code>tr_string_node_t</code> ” on page 3-8 for more information.)

### RETURN VALUES

Returns zero if no errors have occurred (since the last call to `tr_error_clear()` or the beginning of the program); otherwise, returns the number of errors in the list of errors pointed to by *list*. If the user passes in a NULL value for the address of *list*, *list* is not set.

See “Error Detection, Collection, and Reporting” on page 3-15 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “`tr_t`” on page 3-8
- “`tr_string_node_t`” on page 3-8
- “`tr_error_clear()`” on page 3-15

## Input Specification and Streaming Control

The functions related to input specification and streaming control are:

- `tr_open_file()` (see page 3-17)
- `tr_open_stream()` (see page 3-18)
- `tr_close()` (see page 3-19)
- `tr_stream_notify()` (see page 3-20)
- `tr_stream_consume()` (see page 3-21)
- `tr_stream_size()` (see page 3-21)

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### **tr\_open\_file()**

`tr_open_file()` opens the specified NightTrace data file and initializes the API for operation on the contained data set.

#### **NOTE**

Currently, only one input source is allowed per handle (until it is closed via `tr_close()`).

#### **SYNTAX**

```
extern int tr_open_file (tr_t t,
                        char * filename);
```

#### **PARAMETERS**

<i>t</i>	data set handle
<i>filename</i>	the pathname of the NightTrace data file

#### **RETURN VALUES**

Returns zero on success; returns -1 if there is an error opening the data file.

See “Input Specification and Streaming Control” on page 3-17 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8
- “tr\_close()” on page 3-19

## tr\_open\_stream()

tr\_open\_stream() associates the specified file descriptor with a stream of raw trace data. The stream is normally generated by invoking **ntraceud** or **ntracekd** with the **--stream** option and piping **stdout** to the user application's **stdin**. Alternatively, the NightTrace GUI can launch a user application providing **stdin** as the data stream.

## NOTE

Currently, only one input source is allowed per handle (until it is closed via tr\_close()).

## SYNTAX

```
extern int tr_open_stream (tr_t t,
                          int fd,
                          int size,
                          int flags);
```

## PARAMETERS

<i>t</i>	data set handle
<i>fd</i>	file descriptor providing streaming raw data
<i>size</i>	the amount of memory (in bytes) to be used to hold trace data that has been read from the file descriptor but not yet consumed by API calls. When the maximum amount of memory has been reached, the pipe will fill and eventually the corresponding daemon will begin to lose events.  The <i>size</i> can be dynamically modified by using tr_stream_size().
<i>flags</i>	may contain the following value:  TR_STREAM_SAVE - this instructs the API to retain all streamed events in memory even after they have been consumed. By default, for streaming data, once an event has been consumed by an API call, its memory will be (eventually) released and it cannot be referenced subsequently.

## RETURN VALUES

Returns zero on success; returns -1 if there is an error opening the data stream.

See “Input Specification and Streaming Control” on page 3-17 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_stream\_size()” on page 3-21
- “tr\_close()” on page 3-19

## tr\_close()

`tr_close()` closes the specified data set and associated data file or stream file descriptor. In the case of a data stream, if the associated daemon is still running, the daemon will terminate with an error.

### NOTE

Currently, only one input source is allowed per handle (until it is closed via `tr_close()`).

### SYNTAX

```
extern void tr_close (tr_t t);
```

### PARAMETERS

*t*                      data set handle

See “Input Specification and Streaming Control” on page 3-17 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_open\_file()” on page 3-17
- “tr\_open\_stream()” on page 3-18

## tr\_stream\_notify()

tr\_stream\_notify() defines a callback which will occur when a stream event occurs as defined by tr\_stream\_event\_t.

### SYNTAX

```
extern int tr_stream_notify (tr_t t,
                             tr_stream_event_t event,
                             tr_stream_func_t func);
```

### PARAMETERS

<i>t</i>	data set handle
<i>event</i>	can be one of the following:  tr_stream_overflow - When the maximum amount of memory has been used to hold events which have not yet been consumed. The user's callback function may wish to increase the size via tr_stream_size().  tr_stream_stall - A stall occurs when there is an insufficient number of events available to form a segment for consumption. The user's callback function may wish to call tr_stream_consume().
<i>func</i>	callback function

### RETURN VALUES

Returns zero on success; returns -1 if the specified arguments are invalid or there is insufficient memory available to register the callback function.

See "Input Specification and Streaming Control" on page 3-17 for related functions.

See "Functions" on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- "tr\_t" on page 3-8
- "tr\_stream\_event\_t" on page 3-8
- "tr\_stream\_func\_t" on page 3-8
- "tr\_stream\_size()" on page 3-21
- "tr\_stream\_consume()" on page 3-21



**tr\_stream\_consume()**

`tr_stream_consume()` forces a segment to be created with all events that are available. Normally, API calls which consume events will wait until enough events are available to make a full segment. A segmented approach is used for performance reasons; unbridled use of `tr_stream_consume()` will affect performance. This function might be used in a stream callback function to force availability of events for situations where the event generation rate is low.

**SYNTAX**

```
extern int tr_stream_consume (tr_t t);
```

**PARAMETERS**

*t*                    data set handle

**RETURN VALUES**

Returns the number of events in the segment just created.

See “Input Specification and Streaming Control” on page 3-17 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8

**tr\_stream\_size()**

`tr_stream_size()` dynamically changes the memory limit originally specified via `tr_open_stream()`. It controls the amount of memory used to hold events that have been read from the stream file descriptor but have not yet been consumed.

**SYNTAX**

```
extern int tr_stream_size (tr_t t,
                          int size);
```

**PARAMETERS**

*t*                    data set handle

*size*                memory limit associated with streaming events

**RETURN VALUES**

Returns zero on success; returns -1 if the specified size is invalid.

See “Input Specification and Streaming Control” on page 3-17 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_open\_stream()” on page 3-18

## Event Offset Positioning

The functions related to event offset positioning are:

- `tr_next_event()` (see page 3-23)
- `tr_next_event_()` (see page 3-24)
- `tr_prev_event()` (see page 3-24)
- `tr_prev_event_()` (see page 3-25)
- `tr_search()` (see page 3-26)
- `tr_seek()` (see page 3-27)

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### **tr\_next\_event()**

`tr_next_event()` advances the offset to the next consecutive trace event.

#### **SYNTAX**

```
extern tr_offset_t tr_next_event (tr_t t);
```

#### **PARAMETERS**

*t*                      data set handle

#### **RETURN VALUES**

Returns the offset of the trace event or `TR_EOF` if the end of the data set has been reached in which case the current position is after the last trace event in the data set.

See “Event Offset Positioning” on page 3-23 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### **SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

## tr\_next\_event\_()

tr\_next\_event\_() advances to the next consecutive trace event meeting the specified condition in the data set.

### SYNTAX

```
extern tr_offset_t tr_next_event_ (tr_t t,  
                                  tr_cond_t condition);
```

### PARAMETERS

<i>t</i>	data set handle
<i>condition</i>	handle of the desired condition

### RETURN VALUES

Returns the offset of the trace event or TR\_EOF if the end of the data set has been reached in which case the current position is after the last trace event in the data set.

See “Event Offset Positioning” on page 3-23 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_offset\_t” on page 3-5

## tr\_prev\_event()

tr\_prev\_event() advances to the previous trace event.

### SYNTAX

```
extern tr_offset_t tr_prev_event (tr_t t);
```

### PARAMETERS

<i>t</i>	data set handle
----------	-----------------

### RETURN VALUES

Returns the offset of the trace event or TR\_EOF if the end of the data set has been reached in which case the current position is before the first event in the data set.

See “Event Offset Positioning” on page 3-23 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

## tr\_prev\_event\_()

tr\_prev\_event\_() advances to the next consecutive trace event meeting the specified condition in the data set.

### SYNTAX

```
extern tr_offset_t tr_prev_event_ (tr_t t,
                                  tr_cond_t condition);
```

### PARAMETERS

<i>t</i>	data set handle
<i>condition</i>	handle of the desired condition

### RETURN VALUES

Returns the offset of the trace event or TR\_EOF if the end of the data set has been reached in which case the current position is before the first event in the data set.

See “Event Offset Positioning” on page 3-23 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_offset\_t” on page 3-5

## tr\_search()

tr\_search() searches for the trace event matching the specified condition in the direction specified. The current position remains unchanged.

### SYNTAX

```
extern tr_offset_t tr_search(tr_t t,  
                             tr_dir_t direction,  
                             tr_cond_t condition);
```

### PARAMETERS

<i>t</i>	data set handle
<i>direction</i>	direction in which to search
<i>condition</i>	handle of the desired condition

### RETURN VALUES

Returns the position of the matching trace event; if no matching event is found, TR\_EOF is returned.

See “Event Offset Positioning” on page 3-23 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_dir\_t” on page 3-5
- “tr\_cond\_t” on page 3-5
- “tr\_offset\_t” on page 3-5

**tr\_seek()**

`tr_seek()` sets the position to the specified offset. If the offset specifies a position that exceeds the offset of the last trace event, the position is set to the last event in the data set.

**SYNTAX**

```
extern tr_offset_t tr_seek (tr_t t,  
                           tr_offset_t offset);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>offset</i>	offset of the trace event

**RETURN VALUES**

The offset of the trace event at the resultant position is returned.

See “Event Offset Positioning” on page 3-23 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

## Basic Event Attribute Functions

The functions that deal with the basic attributes of trace events are:

- `tr_id()` (see page 3-29)
- `tr_id_()` (see page 3-29)
- `tr_time()` (see page 3-30)
- `tr_time_()` (see page 3-31)
- `tr_nargs()` (see page 3-31)
- `tr_nargs_()` (see page 3-32)
- `tr_arg_int()` (see page 3-33)
- `tr_arg_int_()` (see page 3-33)
- `tr_arg_dbl()` (see page 3-34)
- `tr_arg_dbl_()` (see page 3-35)
- `tr_pid()` (see page 3-35)
- `tr_pid_()` (see page 3-36)
- `tr_raw_pid()` (see page 3-38)
- `tr_raw_pid_()` (see page 3-39)
- `tr_lwpid()` (see page 3-40)
- `tr_lwpid_()` (see page 3-41)
- `tr_tid()` (see page 3-41)
- `tr_tid_()` (see page 3-42)
- `tr_thread_id()` (see page 3-43)
- `tr_thread_id_()` (see page 3-43)
- `tr_task_id()` (see page 3-44)
- `tr_task_id_()` (see page 3-45)
- `tr_cpu()` (see page 3-45)
- `tr_cpu_()` (see page 3-46)
- `tr_node()` (see page 3-47)
- `tr_node_()` (see page 3-48)
- `tr_process_name()` (see page 3-48)
- `tr_process_name_()` (see page 3-49)
- `tr_task_name()` (see page 3-50)
- `tr_task_name_()` (see page 3-50)



- `tr_thread_name()` (see page 3-51)
- `tr_thread_name_()` (see page 3-51)

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## **tr\_id()**

`tr_id()` returns the trace ID associated with the current trace event.

### **SYNTAX**

```
extern int tr_id (tr_t t);
```

### **PARAMETERS**

*t*                      data set handle

### **RETURN VALUES**

Returns the trace ID associated with the current trace event.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### **SEE ALSO**

- “`tr_t`” on page 3-8

## **tr\_id\_()**

`tr_id_()` returns the trace ID associated with the trace event at the specified offset.

### **SYNTAX**

```
extern int tr_id_ (tr_t t,
                  tr_offset_t offset);
```

### **PARAMETERS**

*t*                      data set handle

*offset*                offset of the trace event

## RETURN VALUES

Returns the trace ID associated with the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

## tr\_time()

tr\_time() returns the timestamp (in seconds) of the current trace event.

## NOTE

A timestamp is relative to the beginning of the trace logging daemon.

## SYNTAX

```
extern double tr_time (tr_t t);
```

## PARAMETERS

*t* data set handle

## RETURN VALUES

Returns the timestamp (in seconds) of the current trace event.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8

**tr\_time\_()**

`tr_time_()` returns the timestamp (in seconds) of the trace event at the specified offset.

**NOTE**

A timestamp is relative to the beginning of the trace logging daemon.

**SYNTAX**

```
extern double tr_time_ (tr_t t,
                       tr_offset_t offset);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>offset</i>	offset of the trace event

**RETURN VALUES**

Returns the timestamp (in seconds) of the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

**tr\_nargs()**

`tr_nargs()` returns the number of arguments associated with the current trace event.

**SYNTAX**

```
extern int tr_nargs (tr_t t);
```

**PARAMETERS**

<i>t</i>	data set handle
----------	-----------------

## RETURN VALUES

Returns the number of arguments associated with the current trace event.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8

## tr\_nargs\_()

tr\_nargs\_() returns the number of arguments associated with the trace event at the specified offset.

## SYNTAX

```
extern int tr_nargs_ (tr_t t,  
                    tr_offset_t offset);
```

## PARAMETERS

<i>t</i>	data set handle
<i>offset</i>	offset of the trace event

## RETURN VALUES

Returns the number of arguments associated with the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

**tr\_arg\_int()**

`tr_arg_int()` returns the desired integer argument of the current trace event.

**SYNTAX**

```
extern int tr_arg_int (tr_t t,
                      int arg_number);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>arg_number</i>	number of the desired argument

**RETURN VALUES**

Returns the desired integer argument of the current trace event; returns zero if an invalid offset is specified or an invalid argument number is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8

**tr\_arg\_int\_()**

`tr_arg_int_()` returns the desired integer argument of the trace event at the specified offset.

**SYNTAX**

```
extern int tr_arg_int_ (tr_t t,
                       int arg_number,
                       tr_offset_t offset);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>arg_number</i>	number of the desired argument
<i>offset</i>	offset of the trace event

## RETURN VALUES

Returns the desired integer argument of the trace event at the specified offset; returns zero if an invalid offset is specified or an invalid argument number is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

## tr\_arg\_dbl()

tr\_arg\_dbl() returns the desired double argument of the current trace event.

## SYNTAX

```
extern double tr_arg_dbl (tr_t t,  
                        int arg_number);
```

## PARAMETERS

<i>t</i>	data set handle
<i>arg_number</i>	number of the desired argument

## RETURN VALUES

Returns the desired double argument of the current trace event; returns zero if an invalid offset is specified or an invalid argument number is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8

**tr\_arg\_dbl\_()**

`tr_arg_dbl_()` returns the desired double argument of the trace event at the specified offset.

**SYNTAX**

```
extern double tr_arg_dbl_ (tr_t t,
                          int arg_number,
                          tr_offset_t offset);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>arg_number</i>	number of the desired argument
<i>offset</i>	offset of the trace event

**RETURN VALUES**

Returns the desired double argument of the trace event at the specified offset; returns zero if an invalid offset is specified or an invalid argument number is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

**tr\_pid()**

`tr_pid()` returns the global process identifier (*PID*) associated with the current trace event.

**NOTE**

On PowerMAX OS systems, a global process identifier does not have the same meaning as the typical operating system definition of `pid`. A PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in

the lower 16 bits. Consult the `_lwp_global_self(2)` man page for more information.

On Linux systems, `tr_pid()` is the same as the operating system process identifier.

## SYNTAX

```
extern int tr_pid (tr_t t);
```

## PARAMETERS

*t*                      data set handle

## RETURN VALUES

Returns the process ID of the current trace event.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8

## tr\_pid\_()

`tr_pid_()` returns the global process identifier (*PID*) associated with the trace event at the specified offset.

## NOTE

On PowerMAX OS systems, a global process identifier does not have the same meaning as the typical operating system definition of **pid**. A PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in



the lower 16 bits. Consult the `_lwp_global_self(2)` man page for more information.

On Linux systems, the `tr_pid_()` is the same as the operating system process identifier.

## SYNTAX

```
extern int tr_pid_ (tr_t t,
                  tr_offset_t offset);
```

## PARAMETERS

<i>t</i>	data set handle
<i>offset</i>	offset of the trace event

## RETURN VALUES

Returns the global process identifier (*PID*) associated with the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

## tr\_raw\_pid()

tr\_raw\_pid() returns the process identifier (*raw PID*) associated with the current trace event.

### NOTE

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. The tr\_raw\_pid() function returns the upper 16 bits.

On Linux systems, the tr\_raw\_pid() is the same as the operating system process identifier.

### SYNTAX

```
extern int tr_raw_pid (tr_t t);
```

### PARAMETERS

*t* data set handle

### RETURN VALUES

Returns the process identifier (*raw PID*) associated with the current trace event.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8

**tr\_raw\_pid\_()**

`tr_raw_pid_()` returns the process identifier (*raw PID*) associated with the trace event at the specified offset.

**NOTE**

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. The `tr_raw_pid_()` function returns the upper 16 bits.

On Linux systems, the `tr_raw_pid_()` is the same as the operating system process identifier.

**SYNTAX**

```
extern int tr_raw_pid_ (tr_t t,
                       tr_offset_t offset);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>offset</i>	offset of the trace event

**RETURN VALUES**

Returns the process identifier (*raw PID*) associated with the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

## tr\_lwpid()

tr\_lwpid() returns the lightweight process identifier (*LWPID*) associated with the current trace event.

### NOTE

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. The tr\_lwpid() function returns the lower 16 bits. See the `_lwp_self(2)` man page for more information.

On Linux systems, the lwpid() is the same as the operating system process identifier.

### SYNTAX

```
extern int tr_lwpid (tr_t t);
```

### PARAMETERS

*t*                      data set handle

### RETURN VALUES

Returns the lightweight process identifier (*LWPID*) associated with the current trace event.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8

**tr\_lwpid\_()**

`tr_lwpid_()` returns the lightweight process identifier (*LWPID*) associated with the trace event at the specified offset.

**SYNTAX**

```
extern int tr_lwpid_ (tr_t t,
                    tr_offset_t offset);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>offset</i>	offset of the trace event

**RETURN VALUES**

Returns the lightweight process identifier (*LWPID*) associated with the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

**tr\_tid()**

`tr_tid()` returns the internally-assigned NightTrace thread identifier (*TID*) associated with the current trace event.

**SYNTAX**

```
extern int tr_tid (tr_t t);
```

**PARAMETERS**

<i>t</i>	data set handle
----------	-----------------

**RETURN VALUES**

Returns the internally-assigned NightTrace thread identifier (*TID*) associated with the current trace event.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### SEE ALSO

- “tr\_t” on page 3-8

### tr\_tid\_()

tr\_tid\_() returns the internally-assigned NightTrace thread identifier (*TID*) associated with the trace event at the specified offset.

#### SYNTAX

```
extern int tr_tid_ (tr_t t,  
                  tr_offset_t offset);
```

#### PARAMETERS

<i>t</i>	data set handle
<i>offset</i>	offset of the trace event

#### RETURN VALUES

Returns the internally-assigned NightTrace thread identifier (*TID*) associated with the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

**tr\_thread\_id()**

`tr_thread_id()` returns the *thread* identifier associated with the current trace event.

**NOTE**

See the **thr\_self(3thread)** man page for more information.

**SYNTAX**

```
extern int tr_thread_id (tr_t t);
```

**PARAMETERS**

*t* data set handle

**RETURN VALUES**

Returns the thread identifier associated with the current trace event.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8

**tr\_thread\_id\_()**

`tr_thread_id_()` returns the *thread* identifier associated with the trace event at the specified offset.

**NOTE**

See the **thr\_self(3thread)** man page for more information.

**SYNTAX**

```
extern int tr_thread_id_ (tr_t t,
                          tr_offset_t offset);
```

**PARAMETERS**

*t* data set handle

*offset*                      offset of the trace event

### RETURN VALUES

Returns the thread identifier associated with the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

### tr\_task\_id()

tr\_task\_id() returns the Ada task identifier associated with the current trace event.

### NOTE

This function is only meaningful for trace events logged by Ada tasking programs.

### SYNTAX

```
extern int tr_task_id (tr_t t);
```

### PARAMETERS

*t*                              data set handle

### RETURN VALUES

Returns the Ada task identifier associated with the current trace event; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.



**SEE ALSO**

- “tr\_t” on page 3-8

**tr\_task\_id\_()**

`tr_task_id_()` returns the Ada task identifier associated with the trace event at the specified offset.

**SYNTAX**

```
extern int tr_task_id_ (tr_t t,
                      tr_offset_t offset);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>offset</i>	offset of the trace event

**RETURN VALUES**

Returns the Ada task identifier associated with the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

**tr\_cpu()**

`tr_cpu()` returns the CPU where the current trace event was logged. CPUs are logically numbered starting at 0 and monotonically increase thereafter.

## NOTE

The CPU is only recorded for trace events logged by the operating system kernel.

## SYNTAX

```
extern int tr_cpu (tr_t t);
```

## PARAMETERS

*t* data set handle

## RETURN VALUES

Returns the CPU where the current trace event was logged. For trace events not logged by the operating system kernel, a value of -1 is returned (which indicates any CPU).

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8

## tr\_cpu\_()

tr\_cpu\_() returns the CPU where the current trace event was logged. CPUs are logically numbered starting at 0 and monotonically increase thereafter.

## NOTE

The CPU is only recorded for trace events logged by the operating system kernel.

## SYNTAX

```
extern int tr_cpu_ (tr_t t,  
                  tr_offset_t offset);
```

## PARAMETERS

*t* data set handle

*offset* offset of the trace event

## RETURN VALUES

Returns the CPU where the current trace event was logged. For trace events not logged by the operating system kernel, a value of -1 is returned (which indicates any CPU). If an invalid offset is specified, a value of -1 is returned.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

## tr\_node()

tr\_node() returns the name of the system where the current trace event was logged.

## SYNTAX

```
extern char * tr_node (tr_t t);
```

## PARAMETERS

*t* data set handle

## RETURN VALUES

Returns the name of the system where the current trace event was logged.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8

## tr\_node\_()

tr\_node\_() returns the name of the system where the trace event at the specified offset was logged.

### SYNTAX

```
extern char * tr_node_ (tr_t t,  
                       tr_offset_t offset);
```

### PARAMETERS

<i>t</i>	data set handle
<i>offset</i>	offset of the trace event

### RETURN VALUES

Returns the name of the system where the trace event at the specified offset was logged; returns NULL if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

## tr\_process\_name()

tr\_process\_name() returns the name of the process associated with the current trace event.

### SYNTAX

```
extern char * tr_process_name (tr_t t);
```

### PARAMETERS

<i>t</i>	data set handle
----------	-----------------

### RETURN VALUES

Returns the name of the process associated with the current trace event.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8

## tr\_process\_name\_()

tr\_process\_name\_() returns the name of the process associated with the trace event at the specified offset.

### SYNTAX

```
extern char * tr_process_name_ (tr_t t,
                               tr_offset_t offset);
```

### PARAMETERS

<i>t</i>	data set handle
<i>offset</i>	offset of the trace event

### RETURN VALUES

Returns the name of the process associated with the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

## **tr\_task\_name()**

`tr_task_name()` returns the name of the task associated with the current trace event.

### **SYNTAX**

```
extern char * tr_task_name (tr_t t);
```

### **PARAMETERS**

*t*                      data set handle

### **RETURN VALUES**

Returns the name of the task associated with the current trace event.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### **SEE ALSO**

- “tr\_t” on page 3-8

## **tr\_task\_name\_()**

`tr_task_name_()` returns the name of the task associated with the trace event at the specified offset.

### **SYNTAX**

```
extern char * tr_task_name_ (tr_t t,  
                             tr_offset_t offset);
```

### **PARAMETERS**

*t*                      data set handle

*offset*                offset of the trace event

### **RETURN VALUES**

Returns the name of the task associated with the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5

### tr\_thread\_name()

tr\_thread\_name() returns the thread name associated with the current trace event.

#### SYNTAX

```
extern char * tr_thread_name (tr_t t);
```

#### PARAMETERS

*t* data set handle

#### RETURN VALUES

Returns the thread name associated with the current trace event.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### SEE ALSO

- “tr\_t” on page 3-8

### tr\_thread\_name\_()

tr\_thread\_name\_() returns the thread name associated with the trace event at the specified offset.

#### SYNTAX

```
extern char * tr_thread_name_ (tr_t t,
                               tr_offset_t offset);
```

#### PARAMETERS

*t* data set handle  
*offset* offset of the trace event

## **RETURN VALUES**

Returns the thread name associated with the trace event at the specified offset; returns zero if an invalid offset is specified.

See “Basic Event Attribute Functions” on page 3-28 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## **SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_offset\_t” on page 3-5



## Conditions

The functions that deal with the creation and manipulation of conditions and their requirements are:

- `tr_cond_create()` (see page 3-54)
- `tr_cond_reset()` (see page 3-55)
- `tr_cond_find()` (see page 3-55)
- `tr_cond_id()` (see page 3-56)
- `tr_cond_id_range()` (see page 3-57)
- `tr_cond_id_clear()` (see page 3-58)
- `tr_cond_cpu()` (see page 3-59)
- `tr_cond_cpu_clear()` (see page 3-59)
- `tr_cond_pid()` (see page 3-60)
- `tr_cond_pid_name()` (see page 3-61)
- `tr_cond_pid_clear()` (see page 3-62)
- `tr_cond_tid()` (see page 3-63)
- `tr_cond_tid_name()` (see page 3-64)
- `tr_cond_tid_clear()` (see page 3-65)
- `tr_cond_node()` (see page 3-66)
- `tr_cond_node_clear()` (see page 3-67)
- `tr_cond_func_or()` (see page 3-68)
- `tr_cond_func_and()` (see page 3-70)
- `tr_cond_func_clear()` (see page 3-72)
- `tr_cond_expr_and()` (see page 3-73)
- `tr_cond_expr_or()` (see page 3-74)
- `tr_cond_not()` (see page 3-75)
- `tr_cond_or()` (see page 3-76)
- `tr_cond_and()` (see page 3-77)
- `tr_cond_copy()` (see page 3-78)
- `tr_cond_name()` (see page 3-79)
- `tr_cond_satisfy()` (see page 3-79)
- `tr_cond_satisfy_()` (see page 3-80)
- `tr_cond_register()` (see page 3-81)

- `tr_cond_offset()` (see page 3-82)

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## **tr\_cond\_create()**

`tr_cond_create()` creates a new condition which will (initially) match all events.

### **SYNTAX**

```
extern tr_cond_t tr_cond_create (tr_t t,  
                                char * name);
```

### **PARAMETERS**

<i>t</i>	data set handle
<i>name</i>	name to subsequently reference newly-created condition; if the name is non-null, the condition may be retrieved via <code>tr_cond_find()</code> subsequently; if a condition with the same name already exists, the existing condition will become unnamed but will not be otherwise modified.

### **RETURN VALUES**

Returns an opaque handle which identifies the condition; in the event there is insufficient memory to create the condition, `TR_NO_COND` will be returned.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### **SEE ALSO**

- “`tr_t`” on page 3-8
- “`tr_cond_t`” on page 3-5
- “`tr_cond_find()`” on page 3-55

**tr\_cond\_reset()**

`tr_cond_reset()` resets the condition to match all events; all previous modifications to the specified condition are discarded.

**SYNTAX**

```
extern void tr_cond_reset (tr_t t,
                          tr_cond_t cond);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>cond</i>	handle of condition to reset

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_create()” on page 3-54

**tr\_cond\_find()**

`tr_cond_find()` locates an existing condition (perhaps imported from a file) and returns its handle.

**SYNTAX**

```
extern tr_cond_t tr_cond_find (tr_t t,
                              char * name);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>name</i>	name used to reference the desired condition as defined in <code>tr_cond_create()</code>

**RETURN VALUES**

Returns the handle of the desired condition; returns `TR_NO_COND` if the named condition does not exist.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_create()” on page 3-54

### tr\_cond\_id()

tr\_cond\_id() appends the specified trace ID to the list of required trace IDs that must be matched for a particular condition to evaluate to TRUE.

### NOTE

Before the first tr\_cond\_id() or tr\_cond\_id\_range() call, or after calling tr\_cond\_id\_clear(), the trace ID requirement is empty which matches any ID.

### SYNTAX

```
extern int tr_cond_id (tr_t t,  
                      tr_cond_t cond,  
                      int id);
```

### PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition with which the given trace ID is to be associated
<i>id</i>	trace ID to add to those that must be matched for the given condition to evaluate to TRUE

### RETURN VALUES

Returns zero on success or non-zero if insufficient memory is available to register the ID.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_create()” on page 3-54
- “tr\_cond\_id\_range()” on page 3-57
- “tr\_cond\_id\_clear()” on page 3-58

## tr\_cond\_id\_range()

`tr_cond_id_range()` appends the trace IDs included in the given trace ID range to the list of required trace IDs that must be matched for the given condition to evaluate to TRUE.

## NOTE

Before the first `tr_cond_id()` or `tr_cond_id_range()` call, or after calling `tr_cond_id_clear()`, the trace ID requirement is empty which matches any ID.

## SYNTAX

```
extern int tr_cond_id_range (tr_t t,
                           tr_cond_t cond,
                           int id1,
                           int id2);
```

## PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition with which the given trace ID range is to be associated
<i>id1</i>	minimum value in the range of trace IDs to be associated with the given condition
<i>id2</i>	maximum value in the range of trace IDs to be associated with the given condition

## RETURN VALUES

Returns zero on success or non-zero if insufficient memory is available to register the IDs.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_id()” on page 3-56
- “tr\_cond\_id\_clear()” on page 3-58

### tr\_cond\_id\_clear()

`tr_cond_id_clear()` removes all trace ID requirements from a particular condition.

### NOTE

Before the first `tr_cond_id()` or `tr_cond_id_range()` call, or after calling `tr_cond_id_clear()`, the trace ID requirement is empty which matches any ID.

### SYNTAX

```
extern void tr_cond_id_clear (tr_t t,  
                             tr_cond_t cond);
```

### PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition from which all trace ID requirements will be removed

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_id()” on page 3-56
- “tr\_cond\_id\_range()” on page 3-57

**tr\_cond\_cpu()**

`tr_cond_cpu()` sets the CPU requirement to any of the CPUs defined in the specified CPU bias.

**SYNTAX**

```
extern void tr_cond_cpu (tr_t t,
                        tr_cond_t cond,
                        int cpu_bias);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>cond</i>	handle of the condition with which to associate the given CPU bias
<i>cpu_bias</i>	CPU bias to apply to the given condition

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_cpu\_clear()” on page 3-59

**tr\_cond\_cpu\_clear()**

`tr_cond_cpu_clear()` clears the CPU requirement for the given condition.

**NOTE**

This function is equivalent to calling `tr_cond_cpu()` with -1 as the CPU bias.

**SYNTAX**

```
extern void tr_cond_cpu_clear (tr_t t,
                              tr_cond_t cond);
```

**PARAMETERS**

<i>t</i>	data set handle
----------	-----------------

*cond* handle of the condition

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_cpu()” on page 3-59

### tr\_cond\_pid()

tr\_cond\_pid() appends the specified process ID to the list of required processes that must be matched for the given condition to evaluate to TRUE.

### NOTE

Before the first tr\_cond\_pid() call or tr\_cond\_pid\_name(), or after calling tr\_cond\_pid\_clear(), the process requirement is empty which matches any process.

### SYNTAX

```
extern int tr_cond_pid (tr_t t,  
                      tr_cond_t cond,  
                      int pid);
```

### PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition
<i>pid</i>	process ID to be added to the list of processes associated with the given condition

### RETURN VALUES

Returns zero on success or non-zero if insufficient memory is available to register the specified process ID.

See “Conditions” on page 3-53 for related functions.



See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_pid\_name()” on page 3-61
- “tr\_cond\_pid\_clear()” on page 3-62

### tr\_cond\_pid\_name()

`tr_cond_pid_name()` appends the process with the specified name to the list of required processes that must be matched for the given condition to evaluate to `TRUE`.

### NOTE

Before the first `tr_cond_pid()` call or `tr_cond_pid_name()`, or after calling `tr_cond_pid_clear()`, the process requirement is empty which matches any process.

### SYNTAX

```
extern int tr_cond_pid_name (tr_t t,
                           tr_cond_t cond,
                           char * process_name);
```

### PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition
<i>process_name</i>	name of the process to be added to the list of processes associated with the given condition

### RETURN VALUES

Returns zero on success or non-zero if insufficient memory is available to register the process with the specified name.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_pid()” on page 3-60
- “tr\_cond\_pid\_clear()” on page 3-62

**tr\_cond\_pid\_clear()**

tr\_cond\_pid\_clear() removes all process requirements from a particular condition.

**NOTE**

Before the first tr\_cond\_pid() call or tr\_cond\_pid\_name(), or after calling tr\_cond\_pid\_clear(), the process requirement is empty which matches any process.

**SYNTAX**

```
extern void tr_cond_pid_clear (tr_t t,  
                             tr_cond_t cond);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>cond</i>	handle of the condition

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_pid()” on page 3-60
- “tr\_cond\_pid\_name()” on page 3-61

**tr\_cond\_tid()**

`tr_cond_tid()` appends the specified thread ID to the list of required threads IDs that must be matched for the given condition to evaluate to TRUE.

**NOTE**

Before the first `tr_cond_tid()` call or `tr_cond_tid_name()`, or after calling `tr_cond_tid_clear()`, the thread requirement is empty which matches any thread.

**SYNTAX**

```
extern int tr_cond_tid (tr_t t,
                      tr_cond_t cond,
                      int tid);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>cond</i>	handle of the condition
<i>tid</i>	thread ID to be added to the list of threads associated with the given condition

**RETURN VALUES**

Returns zero on success or non-zero if insufficient memory is available to register the specified thread ID.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_tid\_name()” on page 3-64
- “tr\_cond\_tid\_clear()” on page 3-65

## tr\_cond\_tid\_name()

tr\_cond\_tid\_name() appends the thread with the specified name to the list of required threads that must be matched for the given condition to evaluate to TRUE.

### NOTE

Before the first tr\_cond\_tid() call or tr\_cond\_tid\_name(), or after calling tr\_cond\_tid\_clear(), the thread requirement is empty which matches any thread.

### SYNTAX

```
extern int tr_cond_tid_name (tr_t t,
                             tr_cond_t cond,
                             char * tid_name);
```

### PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition
<i>tid_name</i>	name of the thread to be added to the list of threads associated with the given condition

### RETURN VALUES

Returns zero on success or non-zero if insufficient memory is available to register the thread with the specified name.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_tid()” on page 3-63
- “tr\_cond\_tid\_clear()” on page 3-65

**tr\_cond\_tid\_clear()**

`tr_cond_tid_clear()` removes all thread requirements from a particular condition.

**NOTE**

Before the first `tr_cond_tid()` call or `tr_cond_tid_name()`, or after calling `tr_cond_tid_clear()`, the thread requirement is empty which matches any thread.

**SYNTAX**

```
extern void tr_cond_tid_clear (tr_t t,
                              tr_cond_t cond);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>cond</i>	handle of the condition

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5

## tr\_cond\_node()

tr\_cond\_node() appends the specified system node name to the list of required node names that must be matched for the given condition to evaluate to TRUE.

### NOTE

Before the first tr\_cond\_node() call or after calling tr\_cond\_node\_clear(), the node requirement is empty which matches any node.

### SYNTAX

```
extern int tr_cond_node (tr_t t,  
                        tr_cond_t cond,  
                        char * node);
```

### PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition
<i>node</i>	name of the node to be added to the list of nodes associated with the given condition

### RETURN VALUES

Returns zero on success or non-zero if insufficient memory is available to register the specified node.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_node\_clear()” on page 3-67

**tr\_cond\_node\_clear()**

`tr_cond_node_clear()` removes all node name requirements from a particular condition.

**NOTE**

Before the first `tr_cond_node()` call or after calling `tr_cond_node_clear()`, the node requirement is empty which matches any node.

**SYNTAX**

```
extern void tr_cond_node_clear (tr_t t,
                               tr_cond_t cond);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>cond</i>	handle of the condition

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_node()” on page 3-66

## tr\_cond\_func\_or()

tr\_cond\_func\_or() modifies the specified condition to include an additional requirement as specified by the user-callable function. The context parameter will be passed to the specified user function.

### NOTE

Multiple requirements may be appended by calling tr\_cond\_or() / tr\_cond\_and() multiple times on the same condition.

### SYNTAX

```
extern int tr_cond_func_or (tr_t t,
                           tr_cond_t cond,
                           tr_cond_func_t func,
                           void *context);
```

### PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition
<i>func</i>	user-callable function to be associated with the given condition
<i>context</i>	

### ADDITIONAL INFORMATION

When the API evaluates the condition, it first ensures that the following requirements (if they exist) are met:

- event's trace ID matches or is within any specified trace ID or trace ID range
- event's process ID matches one of the specified process IDs
- event's thread ID matches one of the specified thread IDs
- event's task ID matches one of the specified task IDs
- event's node name matches one of the specified node names
- event's CPU intersects the specified CPU bias

If and only if these requirements are met, then the user's function is called.

The user function should return 1 (true) if the user's requirement is met or 0 (false) if it is not met.

Before calling tr\_cond\_func\_or(), the condition will evaluate to TRUE if all other requirements have been met.



User-defined functions may not be called by the API if the initial requirements are not met or if the left hand side of short circuit boolean condition already resolves the condition.

User-defined functions are invoked in reverse order from which they are specified with the following parenthetical relationship:

*last\_function* **OPERATOR** (*previous\_function*)

Thus calling:

```
tr_cond_func_or(cond,A);
tr_cond_func_and(cond,B);
tr_cond_func_or(cond,C);
tr_cond_func_and(cond,D);
```

would result in the following evaluation:

```
return D && (C || (B && A))
```

## RETURN VALUES

Returns zero on success and non-zero if insufficient memory is available to register the user function with the specified condition.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_func\_t” on page 3-4
- “tr\_cond\_or()” on page 3-76
- “tr\_cond\_and()” on page 3-77
- “tr\_cond\_func\_and()” on page 3-70
- “tr\_cond\_func\_clear()” on page 3-72

## tr\_cond\_func\_and()

`tr_cond_func_and()` modifies the specified condition to include an additional requirement as specified by the user-callable function. The context parameter will be passed to the specified user function.

### NOTE

Multiple requirements may be appended by calling `tr_cond_or()` / `tr_cond_and()` multiple times on the same condition.

### SYNTAX

```
extern int tr_cond_func_and (tr_t t,
                             tr_cond_t cond,
                             tr_cond_func_t func,
                             void *context);
```

### PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition
<i>func</i>	user-callable function to be associated with the given condition
<i>context</i>	

### ADDITIONAL INFORMATION

When the API evaluates the condition, it first ensures that the following requirements (if they exist) are met:

- event's trace ID matches or is within any specified trace ID or trace ID range
- event's process ID matches one of the specified process IDs
- event's thread ID matches one of the specified thread IDs
- event's task ID matches one of the specified task IDs
- event's node name matches one of the specified node names
- event's CPU intersects the specified CPU bias

If and only if these requirements are met, then the user's function is called.

The user function should return 1 (true) if the user's requirement is met or 0 (false) if it is not met.

Before calling `tr_cond_func_and()`, the condition will evaluate to `TRUE` if all other requirements have been met.

User-defined functions may not be called by the API if the initial requirements are not met or if the left hand side of short circuit boolean condition already resolves the condition.

User-defined functions are invoked in reverse order from which they are specified with the following parenthetical relationship:

*last\_function* **OPERATOR** (*previous\_function*)

Thus calling:

```
tr_cond_func_or(cond,A);
tr_cond_func_and(cond,B);
tr_cond_func_or(cond,C);
tr_cond_func_and(cond,D);
```

would result in the following evaluation:

```
return D && (C || (B && A))
```

## RETURN VALUES

Returns zero on success and non-zero if insufficient memory is available to register the user function with the specified condition.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_func\_t” on page 3-4
- “tr\_cond\_or()” on page 3-76
- “tr\_cond\_and()” on page 3-77
- “tr\_cond\_func\_or()” on page 3-68
- “tr\_cond\_func\_and()” on page 3-70

## **tr\_cond\_func\_clear()**

`tr_cond_func_clear()` clears all previously specified user function requirements.

### **SYNTAX**

```
extern void tr_cond_func_clear (tr_t t,  
                               tr_cond_t cond);
```

### **PARAMETERS**

<i>t</i>	data set handle
<i>cond</i>	handle of the condition

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### **SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_func\_or()” on page 3-68
- “tr\_cond\_func\_clear()” on page 3-72

**tr\_cond\_expr\_and()**

`tr_cond_expr_and()` modifies the specified condition to include an additional requirement as specified by a valid NightTrace expression.

**NOTE**

Multiple requirements may be appended by calling `tr_cond_or()` / `tr_cond_and()` multiple times on the same condition.

**SYNTAX**

```
extern char * tr_cond_expr_and (tr_t t,
                               tr_cond_t cond,
                               char * expr);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>cond</i>	handle of the condition
<i>expr</i>	string containing the NightTrace expression to be associated with the given condition

**RETURN VALUES**

Returns zero on success or a character string describing why the specified expression is invalid.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_expr\_or()” on page 3-74

## tr\_cond\_expr\_or()

tr\_cond\_expr\_or() modifies the specified condition to include an additional requirement as specified by a valid NightTrace expression.

### NOTE

Multiple requirements may be appended by calling tr\_cond\_or() / tr\_cond\_and() multiple times on the same condition.

### SYNTAX

```
extern char * tr_cond_expr_or (tr_t t,  
                             tr_cond_t cond,  
                             char * expr);
```

### PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition
<i>expr</i>	string containing the NightTrace expression to be associated with the given condition

### RETURN VALUES

Returns zero on success or a character string describing why the specified expression is invalid.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_expr\_and()” on page 3-73

**tr\_cond\_not()**

`tr_cond_not()` creates a new condition which evaluates to `TRUE` only if the specified condition evaluates to `FALSE`.

**NOTE**

The new condition will still reference the specified condition; thus subsequent changes to the specified condition will affect the outcome of the created condition.

**SYNTAX**

```
extern tr_cond_t tr_cond_not (tr_t t,
                             char* name,
                             tr_cond_t cond);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>name</i>	name to reference the newly-created condition; if an existing condition already exists with the specified <i>name</i> , it becomes unnamed but remains otherwise unchanged; if <i>name</i> is <code>NULL</code> , the newly-created condition will be unnamed
<i>cond</i>	existing condition on which to base the newly-created condition

**RETURN VALUES**

Returns the handle of the newly-created condition; returns `TR_NO_COND` if insufficient memory is available to create the new condition.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_or()” on page 3-76
- “tr\_cond\_and()” on page 3-77

## tr\_cond\_or()

tr\_cond\_or() creates a new condition which evaluates to TRUE if either of the specified conditions evaluate to TRUE.

### NOTE

The new condition will still reference the specified conditions; thus subsequent changes to the specified conditions will affect the outcome of the created condition.

### SYNTAX

```
extern tr_cond_t tr_cond_or (tr_t t,  
                             char * name,  
                             tr_cond_t left,  
                             tr_cond_t right);
```

### PARAMETERS

<i>t</i>	data set handle
<i>name</i>	name to reference the newly-created condition; if an existing condition already exists with the specified <i>name</i> , it becomes unnamed but remains otherwise unchanged; if <i>name</i> is NULL, the newly-created condition will be unnamed
<i>left</i>	one of two existing conditions either of which must evaluate to TRUE for the newly-created condition to evaluate to TRUE
<i>right</i>	one of two existing conditions either of which must evaluate to TRUE for the newly-created condition to evaluate to TRUE

### RETURN VALUES

Returns the handle of the newly-created condition; returns TR\_NO\_COND if insufficient memory is available to create the new condition.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_not()” on page 3-75



- “tr\_cond\_and()” on page 3-77

## tr\_cond\_and()

tr\_cond\_and() creates a new condition which evaluates to TRUE only if both of the specified conditions evaluate to TRUE.

### NOTE

The new condition will still reference the specified conditions; thus subsequent changes to the specified conditions will affect the outcome of the created condition.

### SYNTAX

```
extern tr_cond_t tr_cond_and (tr_t t,
                             char * name,
                             tr_cond_t left,
                             tr_cond_t right);
```

### PARAMETERS

<i>t</i>	data set handle
<i>name</i>	name to reference the newly-created condition; if an existing condition already exists with the specified <i>name</i> , it becomes unnamed but remains otherwise unchanged; if <i>name</i> is NULL, the newly-created condition will be unnamed
<i>left</i>	one of two existing conditions which must both evaluate to TRUE for the newly-created condition to evaluate to TRUE
<i>right</i>	one of two existing conditions which must both evaluate to TRUE for the newly-created condition to evaluate to TRUE

### RETURN VALUES

Returns the handle of the newly-created condition; returns TR\_NO\_COND if insufficient memory is available to create the new condition.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5

- “tr\_cond\_not()” on page 3-75
- “tr\_cond\_or()” on page 3-76

## tr\_cond\_copy()

tr\_cond\_copy() creates a copy of the root of specified condition.

### NOTE

If the specified condition contains references to other conditions, (e.g. it was created by a tr\_cond\_or() / tr\_cond\_and() call), the references remain (i.e. this operation only copies the root and not all conditions it may reference).

### SYNTAX

```
extern tr_cond_t tr_cond_copy (tr_t t,  
                               char * name,  
                               tr_cond_t cond);
```

### PARAMETERS

<i>t</i>	data set handle
<i>name</i>	name to reference the newly-created condition; if an existing condition already exists with the specified <i>name</i> , it becomes unnamed but remains otherwise unchanged; if <i>name</i> is NULL, the newly-created condition will be unnamed
<i>cond</i>	handle of existing condition to copy to create new condition

### RETURN VALUES

Returns the handle of the newly-created copy of the specified condition; returns TR\_NO\_COND if insufficient memory is available to create the new condition.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_or()” on page 3-76
- “tr\_cond\_and()” on page 3-77

**tr\_cond\_name()**

`tr_cond_name()` returns the name of the specified condition.

**SYNTAX**

```
extern char * tr_cond_name (tr_t t,
                           tr_cond_t cond);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>cond</i>	handle of the condition

**RETURN VALUES**

Returns the name of the specified condition (for debugging purposes) or NULL if it is unnamed.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5

**tr\_cond\_satisfy()**

`tr_cond_satisfy()` is used to determine if the current event satisfies the specified condition.

**SYNTAX**

```
extern int tr_cond_satisfy (tr_t t,
                           tr_cond_t cond);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>cond</i>	handle of the condition

## RETURN VALUES

Returns `TRUE` if the current event satisfies the specified condition; returns `FALSE` otherwise.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “`tr_t`” on page 3-8
- “`tr_cond_t`” on page 3-5

## `tr_cond_satisfy_()`

`tr_cond_satisfy_()` is used to determine if the trace event at the specified offset satisfies the specified condition.

## SYNTAX

```
extern int tr_cond_satisfy_ (tr_t t,  
                             tr_cond_t cond,  
                             tr_offset_t offset);
```

## PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition
<i>offset</i>	offset of the trace event

## RETURN VALUES

Returns `TRUE` if the trace event at the specified offset satisfies the specified condition; returns `FALSE` otherwise.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “`tr_t`” on page 3-8
- “`tr_cond_t`” on page 3-5

- “tr\_offset\_t” on page 3-5

## tr\_cond\_register()

tr\_cond\_register() registers the specified condition so that it is evaluated for every event.

### NOTE

Registration of conditions increases processing time.

### SYNTAX

```
extern void tr_cond_register (tr_t t,
                             tr_cond_t cond);
```

### PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of condition to register

### ADDITIONAL INFORMATION

This is the implementation of NightTrace “qualified events” which are basically conditions that are evaluated as each event is consumed.

tr\_activate() should be called after all desired conditions are registered.

Registering conditions is only necessary if you wish to refer to the offset at which the specified condition was last active.

Failure to call tr\_activate() after registration of conditions will result in erroneous statistics about such conditions.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_activate()” on page 3-93
- “Qualified Events” on page 11-113

## tr\_cond\_offset()

tr\_cond\_offset() returns the offset at which the specified condition last evaluated to TRUE.

### SYNTAX

```
extern tr_offset_t tr_cond_offset (tr_t t,  
                                  tr_cond_t cond);
```

### PARAMETERS

<i>t</i>	data set handle
<i>cond</i>	handle of the condition

### RETURN VALUES

Returns the offset at which the specified condition last evaluated to TRUE; returns TR\_EOF if the condition has not yet evaluated to true up to the current offset.

See “Conditions” on page 3-53 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_offset\_t” on page 3-5

## State-oriented Interfaces

The functions that deal with the creation, configuration, and activation of states are:

- `tr_state_create()` (see page 3-84)
- `tr_state_find()` (see page 3-85)
- `tr_state_name()` (see page 3-85)
- `tr_state_start_id()` (see page 3-86)
- `tr_state_start_id_range()` (see page 3-87)
- `tr_state_start_id_clear()` (see page 3-88)
- `tr_state_end_id()` (see page 3-88)
- `tr_state_end_id_range()` (see page 3-89)
- `tr_state_end_id_clear()` (see page 3-90)
- `tr_state_start_cond()` (see page 3-90)
- `tr_state_start_cond_clear()` (see page 3-91)
- `tr_state_end_cond()` (see page 3-92)
- `tr_state_end_cond_clear()` (see page 3-92)
- `tr_activate()` (see page 3-93)
- `tr_state_info()` (see page 3-94)
- `tr_state_info_()` (see page 3-95)
- `tr_state_active()` (see page 3-96)
- `tr_state_active_()` (see page 3-97)

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## tr\_state\_create()

tr\_state\_create() creates a new state with the following attributes:

Start Events:	ALL
End Events:	ALL
Start Condition:	TRUE
End Condition:	TRUE

### SYNTAX

```
extern tr_state_t tr_state_create (tr_t t,  
                                  char * name);
```

### PARAMETERS

<i>t</i>	data set handle
<i>name</i>	name to reference the newly-created state; if an existing state already exists with the specified <i>name</i> , it becomes unnamed but remains otherwise unchanged; if <i>name</i> is NULL, the newly-created state will be unnamed

### RETURN VALUES

Returns an opaque handle which identifies the newly-created state; returns TR\_NO\_STATE if there is insufficient memory available to create the state.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8



**tr\_state\_find()**

`tr_state_find()` locates an existing state (perhaps imported from a file) and returns its handle.

**SYNTAX**

```
extern tr_state_t tr_state_find (tr_t t,
                                char * name);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>name</i>	name used to reference the desired state as defined in <code>tr_state_create()</code>

**RETURN VALUES**

Returns the handle of the desired state; returns `TR_NO_STATE` if the named state does not exist.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “`tr_t`” on page 3-8
- “`tr_state_t`” on page 3-7
- “`tr_state_create()`” on page 3-84

**tr\_state\_name()**

`tr_state_name()` returns the name of the specified state.

**SYNTAX**

```
extern char * tr_state_name (tr_t t,
                             tr_state_t state);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>state</i>	handle of the state

## RETURN VALUES

Returns the name of the specified state (for debugging purposes) or NULL if the state is unnamed.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7

## tr\_state\_start\_id()

tr\_state\_start\_id() appends the specified trace ID to the list of required trace IDs that must be matched for the start event that defines the state.

## SYNTAX

```
extern int tr_state_start_id (tr_t t,  
                             tr_state_t state,  
                             int id);
```

## PARAMETERS

<i>t</i>	data set handle
<i>state</i>	handle of the state
<i>id</i>	trace ID to add to the list of required trace IDs for the start event that defines the state

## RETURN VALUES

Returns zero on success or non-zero if insufficient memory is available to register the ID.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

## SEE ALSO

- “tr\_t” on page 3-8

- “tr\_state\_t” on page 3-7

### tr\_state\_start\_id\_range()

tr\_state\_start\_id\_range() appends the trace IDs included in the given trace ID range to the list of required trace IDs that must be matched for the start event that defines the state.

#### SYNTAX

```
extern int tr_state_start_id_range (tr_t t,
                                   tr_state_t state,
                                   int id1,
                                   int id2);
```

#### PARAMETERS

<i>t</i>	data set handle
<i>state</i>	handle of the state
<i>id1</i>	minimum value in the range of trace IDs to be associated with the given state
<i>id2</i>	maximum value in the range of trace IDs to be associated with the given state

#### RETURN VALUES

Returns zero on success or non-zero if insufficient memory is available to register the IDs.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7

## **tr\_state\_start\_id\_clear()**

`tr_state_start_id_clear()` removes all trace ID requirements related to the start event that defines a particular state (such that that all events are candidates to start a state).

### **SYNTAX**

```
extern void tr_state_start_id_clear (tr_t t,  
                                     tr_state_t state);
```

### **PARAMETERS**

<i>t</i>	data set handle
<i>state</i>	handle of the state

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### **SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7

## **tr\_state\_end\_id()**

`tr_state_end_id()` appends the specified trace ID to the list of required trace IDs that must be matched for the end event that defines the state.

### **SYNTAX**

```
extern int tr_state_end_id (tr_t t,  
                           tr_state_t state,  
                           int id);
```

### **PARAMETERS**

<i>t</i>	data set handle
<i>state</i>	handle of the state
<i>id</i>	trace ID to add to the list of required trace IDs for the end event that defines the state

### **RETURN VALUES**

Returns zero on success or non-zero if insufficient memory is available to register the ID.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7

### tr\_state\_end\_id\_range()

`tr_state_end_id_range()` appends the trace IDs included in the given trace ID range to the list of required trace IDs that must be matched for the end event that defines the state.

### SYNTAX

```
extern int tr_state_end_id_range (tr_t t,
                                tr_state_t state,
                                int id1,
                                int id2);
```

### PARAMETERS

<i>t</i>	data set handle
<i>state</i>	handle of the state
<i>id1</i>	minimum value in the range of trace IDs to be associated with the given state
<i>id2</i>	maximum value in the range of trace IDs to be associated with the given state

### RETURN VALUES

Returns zero on success or non-zero if insufficient memory is available to register the IDs.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8

- “tr\_state\_t” on page 3-7

### tr\_state\_end\_id\_clear()

tr\_state\_end\_id\_clear() removes all trace ID requirements related to the end event that defines a particular state (such that that all events are candidates to end a state).

#### SYNTAX

```
extern void tr_state_end_id_clear (tr_t t,  
                                  tr_state_t state);
```

#### PARAMETERS

<i>t</i>	data set handle
<i>state</i>	handle of the state

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7

### tr\_state\_start\_cond()

tr\_state\_start\_cond() associates a certain condition with start of a particular state.

#### SYNTAX

```
extern void tr_state_start_cond (tr_t t,  
                                tr_state_t state,  
                                tr_cond_t cond);
```

#### PARAMETERS

<i>t</i>	data set handle
<i>state</i>	handle of the state
<i>cond</i>	handle of the condition to associate with the start of the specified state

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7
- “tr\_cond\_t” on page 3-5

### tr\_state\_start\_cond\_clear()

`tr_state_start_cond_clear()` clears any conditions associated with start of a particular state.

### SYNTAX

```
extern void tr_state_start_cond_clear (tr_t t,
                                       tr_state_t state);
```

### PARAMETERS

<i>t</i>	data set handle
<i>state</i>	handle of the state

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7

## **tr\_state\_end\_cond()**

`tr_state_end_cond()` associates a certain condition with end of a particular state.

### **SYNTAX**

```
extern void tr_state_end_cond (tr_t t,  
                             tr_state_t state,  
                             tr_cond_t cond);
```

### **PARAMETERS**

<i>t</i>	data set handle
<i>state</i>	handle of the state
<i>cond</i>	handle of the condition to associate with the end of the specified state

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### **SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7
- “tr\_cond\_t” on page 3-5

## **tr\_state\_end\_cond\_clear()**

`tr_state_end_cond_clear()` clears any conditions associated with end of a particular state.

### **SYNTAX**

```
extern void tr_state_end_cond_clear (tr_t t,  
                                    tr_state_t state);
```

### **PARAMETERS**

<i>t</i>	data set handle
<i>state</i>	handle of the state

See “State-oriented Interfaces” on page 3-83 for related functions.



See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7

## tr\_activate()

`tr_activate()` must be called after the configuration of all states and the registration of all conditions is complete. It may be called multiple times.

### NOTE

Failure to call this function will result in undefined state evaluation and false conditions.

### SYNTAX

```
extern int tr_activate (tr_t t);
```

### PARAMETERS

*t*                      data set handle

### RETURN VALUES

Returns zero upon successful activation or -1 if a circular dependency between states is detected.

### ADDITIONAL INFORMATION

If the current position is other than the beginning of the data set, user-defined functions associated with conditions in states may be called during the invocation of `tr_state_active()`.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_state\_active()” on page 3-96

## tr\_state\_info()

tr\_state\_info() returns a structure containing the current values associated with the last completed instance of the specified state

### SYNTAX

```
extern void tr_state_info (tr_t t,  
                          tr_state_t state,  
                          tr_state_info_t * info);
```

### PARAMETERS

<i>t</i>	data set handle
<i>state</i>	handle of the state
<i>info</i>	pointer to a structure which will contain the current values associated with the last completed instance of the specified state

### RETURN VALUES

The return values are contained in the tr\_state\_info\_t structure (see “tr\_state\_info\_t” on page 3-7).

If the state has never been active, start\_offset and end\_offset are set to TR\_EOF and gap and duration are set to zero.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7
- “tr\_state\_info\_t” on page 3-7

**tr\_state\_info\_()**

`tr_state_info_()` returns a structure containing the current values associated with the given state at the specified offset.

**NOTE**

Calling `tr_state_info_()` is an expensive operation if the specified offset is not the current position.

**SYNTAX**

```
extern void tr_state_info_ (tr_t t,
                          tr_state_t state,
                          tr_state_info_t * info,
                          tr_offset_t offset);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>state</i>	handle of the state
<i>info</i>	pointer to a structure which will contain the current values associated with the given state at the specified offset
<i>offset</i>	offset of the specified state

**RETURN VALUES**

The return values are contained in the `tr_state_info_t` structure (see “`tr_state_info_t`” on page 3-7).

If the state has never been active, `start_offset` and `end_offset` are set to `TR_EOF` and `gap` and `duration` are set to zero.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “`tr_t`” on page 3-8
- “`tr_state_t`” on page 3-7
- “`tr_state_info_t`” on page 3-7
- “`tr_offset_t`” on page 3-5

## **tr\_state\_active()**

`tr_state_active()` is used to determine if the specified state is active at the current offset.

### **SYNTAX**

```
extern int tr_state_active (tr_t t,  
                           tr_state_t state);
```

### **PARAMETERS**

<i>t</i>	data set handle
<i>state</i>	handle of the state

### **RETURN VALUES**

Returns `TRUE` if the specified state is active at the current offset; returns `FALSE` otherwise.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### **SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7

**tr\_state\_active\_()**

`tr_state_active_()` is used to determine if the given state is active at the specified offset.

**NOTE**

Calling `tr_state_active_()` is an expensive operation if the specified offset is not the current position.

**SYNTAX**

```
extern int tr_state_active_ (tr_t t,
                             tr_state_t state,
                             tr_offset_t offset);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>state</i>	handle of the state
<i>offset</i>	offset of the specified state

**RETURN VALUES**

Returns `TRUE` if the given state is active at the specified offset; returns `FALSE` otherwise.

See “State-oriented Interfaces” on page 3-83 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7
- “tr\_offset\_t” on page 3-5

## Output Function

The function dealing with the output of trace data is:

- `tr_copy_input()` (see page 3-98)

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### `tr_copy_input()`

`tr_copy_input()` consumes the entire input data set and copies all events which satisfy the specified condition to the output file.

#### SYNTAX

```
extern int tr_copy_input (tr_t t,
                          char * output_file,
                          tr_cond_t cond,
                          int mode);
```

#### PARAMETERS

<i>t</i>	data set handle
<i>output_file</i>	pathname of the output file
<i>cond</i>	handle of the condition
<i>mode</i>	parameter passed to the system call invoked to open/create the specified output file

#### RETURN VALUES

Returns zero upon success; returns -1 upon error in which case `errno` will be set to a value as per `open(2)` or `read(2)`.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### SEE ALSO

- “`tr_t`” on page 3-8
- “`tr_cond_t`” on page 3-5

## String Table Functions

The following functions are provided to create, manage, and search NightTrace string tables:

- `tr_get_string()` (see page 3-99)
- `tr_get_item()` (see page 3-100)
- `tr_create_table()` (see page 3-101)
- `tr_append_table()` (see page 3-102)

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### `tr_get_string()`

`tr_get_string()` returns the string associated with the number of the desired item in the specified table.

#### SYNTAX

```
extern char * tr_get_string (tr_t t,
                             char * table_name,
                             int item);
```

#### PARAMETERS

<i>t</i>	data set handle
<i>table_name</i>	name of the string table
<i>item</i>	position of the desired item in the specified table

#### RETURN VALUES

Returns the string associated with the number of the desired item in the specified table; returns "" if no match is found.

See “String Table Functions” on page 3-99 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### SEE ALSO

- “`tr_t`” on page 3-8
- “String Tables” on page 4-15

## tr\_get\_item()

tr\_get\_item() returns the item number associated with the string entry in the specified table that matches the specified value.

### SYNTAX

```
extern int tr_get_item (tr_t t,  
                       char * table_name,  
                       char * value);
```

### PARAMETERS

<i>t</i>	data set handle
<i>table_name</i>	name of the table to search for the specified string
<i>value</i>	string entry to search for in the specified table

### RETURN VALUES

Returns the item number associated with the string entry in the specified table that matches the specified value; returns zero if no match is found.

See “String Table Functions” on page 3-99 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “String Tables” on page 4-15



**tr\_create\_table()**

`tr_create_table()` is used to create a string table.

**SYNTAX**

```
extern int tr_create_table (tr_t t,
                           char * table_name,
                           char * default_value,
                           tr_string_node_t * list,
                           int count);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>table_name</i>	name to subsequently reference the newly-created table
<i>default_value</i>	string to associate with integer values that are not explicitly referenced in the table
<i>list</i>	pointer to a list of string table entries
<i>count</i>	number of entries in the list of string table entries

**RETURN VALUES**

Returns zero on success; returns -1 if insufficient memory is available to complete the request or invalid values are specified.

**ADDITIONAL INFORMATION**

All strings referenced by value fields are copied during the operation; therefore the source of the strings need not remain allocated after the call completes.

See “String Table Functions” on page 3-99 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_string\_node\_t” on page 3-8
- “String Tables” on page 4-15

## tr\_append\_table()

tr\_append\_table() associates a particular string with a certain position in a given string table.

### NOTE

If the position specified is already associated with a string, tr\_append\_table() will overwrite the previous entry.

### SYNTAX

```
extern int tr_append_table (tr_t t,
                           char * table_name,
                           char * value,
                           int item);
```

### PARAMETERS

<i>t</i>	data set handle
<i>table_name</i>	name of the table to modify
<i>value</i>	character string to assign to the given item number
<i>item</i>	position in the table to associate with the given string

### RETURN VALUES

Returns zero on success; returns -1 if insufficient memory is available to complete the request or invalid values are specified.

### ADDITIONAL INFORMATION

All strings referenced by value fields are copied during the operation; therefore the source of the strings need not remain allocated after the call completes.

See “String Table Functions” on page 3-99 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “String Tables” on page 4-15

## Callback Interfaces

The following functions deal with the callback capabilities of the NightTrace Analysis Application Programming Interface:

- `tr_iterate()` (see page 3-103)
- `tr_halt()` (see page 3-104)
- `tr_cancel_cb()` (see page 3-104)
- `tr_cond_cb()` (see page 3-105)
- `tr_state_cb()` (see page 3-106)

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### `tr_iterate()`

`tr_iterate()` iteratively processes all events starting at the current position through the end of the data set. For each event, user-defined callback functions registered with `tr_cond_cb()` or `tr_state_cb()` will be invoked as required.

#### SYNTAX

```
extern int tr_iterate (tr_t t);
```

#### PARAMETERS

`t`                      data set handle

#### RETURN VALUES

Returns zero on success and non-zero if an error occurs. Currently, the only error is to reach the memory limit specified on the `tr_open_stream()` call if the input source is streaming data.

See “Callback Interfaces” on page 3-103 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

#### SEE ALSO

- “`tr_t`” on page 3-8
- “`tr_cond_cb()`” on page 3-105
- “`tr_state_cb()`” on page 3-106
- “`tr_open_stream()`” on page 3-18

## tr\_halt()

tr\_halt() halts the iteration process, causing tr\_iterate() to return.

### SYNTAX

```
extern void tr_halt (tr_t t);
```

### PARAMETERS

*t*                      data set handle

See “Callback Interfaces” on page 3-103 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_iterate()” on page 3-103

## tr\_cancel\_cb()

tr\_cancel\_cb() cancels the specified callback.

### SYNTAX

```
extern void tr_cancel_cb (tr_t t,  
                          tr_cb_t cb);
```

### PARAMETERS

*t*                      data set handle

*cb*                     handle of the callback to be cancelled

See “Callback Interfaces” on page 3-103 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_cb\_t” on page 3-3

**tr\_cond\_cb()**

`tr_cond_cb()` registers a user-defined callback function which will be iteratively called for every event that satisfies the specified condition.

**SYNTAX**

```
extern tr_cb_t tr_cond_cb (tr_t t,
                          tr_cond_t cond,
                          tr_cond_cb_func_t func,
                          void * context);
```

**PARAMETERS**

<i>t</i>	data set handle
<i>cond</i>	handle of the condition that must be satisfied in order for the callback function to be called
<i>func</i>	function to be called if the given condition is satisfied for a particular event
<i>context</i>	user defined value which is passed to the specified callback function

**RETURN VALUES**

Returns an opaque handle which identifies the callback; returns `TR_NO_CB` if the specified arguments are invalid or there is insufficient memory available to register the callback function.

See “Callback Interfaces” on page 3-103 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

**SEE ALSO**

- “tr\_t” on page 3-8
- “tr\_cond\_t” on page 3-5
- “tr\_cond\_cb\_func\_t” on page 3-4
- “tr\_cb\_t” on page 3-3

## tr\_state\_cb()

tr\_state\_cb() registers a user-defined callback function which will be iteratively invoked for every event that affects the given state in the manner specified.

### SYNTAX

```
extern tr_cb_t tr_state_cb (tr_t t,  
                           tr_state_t state,  
                           tr_state_action_t action,  
                           tr_state_cb_func_t func,  
                           void * context);
```

### PARAMETERS

<i>t</i>	data set handle
<i>state</i>	handle of the state
<i>action</i>	specifies the manner in which the given function will be called (see “tr_state_action_t” on page 3-6)
<i>func</i>	function which will be iteratively invoked for every event that affects the given state in the specified manner
<i>context</i>	user defined value which is passed to the specified callback function

### RETURN VALUES

Returns an opaque handle which identifies the callback; returns TR\_NO\_CB if the specified arguments are invalid or there is insufficient memory available to register the callback function.

See “Callback Interfaces” on page 3-103 for related functions.

See “Functions” on page 3-9 for a complete list of functions included in the NightTrace Analysis API.

### SEE ALSO

- “tr\_t” on page 3-8
- “tr\_state\_t” on page 3-7
- “tr\_state\_action\_t” on page 3-6
- “tr\_state\_cb\_func\_t” on page 3-6
- “tr\_cb\_t” on page 3-3

## Sample Programs

The following programs are given as examples of how to use the NightTrace Analysis Application Programming Interface (see “NightTrace Analysis Application Programming Interface” on page 3-1).

### NOTE

The source files for these programs are installed in `/usr/lib/NightTrace/examples`.

- **list** (see “list” on page 3-108)

This program simply lists each NightTrace event using a simple main loop to position to the next event.

- **search** (see “search” on page 3-110)

This program utilizes the callback features of the API to locate and describe all events which satisfy a specified condition.

- **watchdog** (see “watchdog” on page 3-112)

This program illustrates how to monitor a certain condition in real-time and then act upon it accordingly.

- **ptime** (see “ptime” on page 3-115)

This program illustrates how to use the NightTrace GUI to export complex conditions and states to a source file which uses the API.

- **browse** (see “browse” on page 3-118)

This program contains a collection of code segments which might be useful for reference.

## list

### Usage

```
./list trace_data_file
```

This program simply lists each NightTrace event using a simple main loop to position to the next event.

See “Sample Programs” on page 3-107 for other programs demonstrating the capabilities of the NightTrace Analysis Application Programming Interface

## list.c

```
#include <unistd.h>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <ntrace_analysis.h>

// Simple example to list all events in a trace data file
// Usage: ./list data_file

static void print (tr_t t, tr_offset_t offset);

int
main (int argc, char * argv[])
{
    tr_t t;
    tr_string_node_t * list;
    tr_offset_t offset;
    int i;
    int errs;

    if (argc != 2) {
        printf ("Usage: list data_file\n");
        exit(1);
    }

    t = tr_init();
    tr_open_file(t,argv[1]);

    errs = tr_error_check(t,&list);
    if (errs) {
        for (i=0; i<errs; ++i)
            printf (" %s (%s)\n", list[i].value, strerror(list[i].item));
        exit(1);
    }

    for (;;) {
```



```
        offset = tr_next_event(t);
        if (offset == TR_EOF) break;
        print(t, offset);
    }

    tr_close(t);
    tr_destroy(&t);
}

static
void
print (tr_t t, tr_offset_t offset)
{
    int i;

    printf ("%5d pid=%5d id=%4d %8.9f nargs=%1d",
            offset,
            tr_pid(t),
            tr_id(t),
            tr_time(t),
            tr_nargs(t));
    for (i=1; i<=tr_nargs(t); ++i) {
        printf (" %5d", tr_arg_int(t,i));
    }
    printf ("\n");
}
```

## search

### Usage

```
./search trace_data_file "NightTrace_Expression"
```

This program utilizes the callback features of the API to locate and describe all events which satisfy the specified condition.

The *NightTrace\_Expression* is a valid NightTrace expression (see “Expressions” on page 11-1) enclosed by double quotes.

The **search** program builds a *condition* object and assigns the specified expression to that condition. It then registers a callback to the `print` function for every event that satisfies the *condition*. It then invokes the `iterate` function to process the entire *trace\_data\_file*.

To call the **search** program with a *trace\_data\_file* named **my\_trace\_data** and the *NightTrace\_Expression*:

```
num_args>1 && arg2==0
```

you would issue the following command:

```
./search my_trace_data "num_args>1 && arg2==0"
```

See “Sample Programs” on page 3-107 for other programs demonstrating the capabilities of the NightTrace Analysis Application Programming Interface

## search.c

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ntrace_analysis.h>

// Simple example to search for all events in a trace data file
// which satisfy the specified condition.

// Usage: ./search data_file "expression"

// Example: ./search data_file "num_args>1 && arg2 == 1"

static void print (tr_t, tr_cond_t c, tr_offset_t, int, void *, int *);

int
main (int argc, char * argv[])
{
    tr_t t;
    tr_string_node_t * list;
```

```

tr_offset_t offset;
tr_cond_t cond;
int i;
int errs;

if (argc < 3) {
    printf ("Usage: search data_file \"expression\"\n");
    exit(1);
}

// Initialize the API and open the input data file
t = tr_init();
tr_open_file(t,argv[1]);

// Create a condition using the specified expression and
// register a callback for it.
cond = tr_cond_create(t,"search");
tr_cond_expr_and(t,cond,argv[2]);
tr_cond_cb(t,cond,print,0);

// Ensure all is copasetic
errs = tr_error_check(t,&list);
if (errs) {
    for (i=0; i<errs; ++i)
        printf ("  %s (%s)\n", list[i].value, strerror(list[i].item));
    exit(1);
}

// Process all events
tr_iterate(t);

tr_close(t);
}

static
void
print (tr_t      t,
       tr_cond_t c,
       tr_offset_t offset,
       int      occurrence,
       void     * context,
       int      * disable)
{
    int i;

    printf ("%5d pid=%5d id=%4d %8.9f nargs=%1d",
            offset,
            tr_pid(t),
            tr_id(t),
            tr_time(t),
            tr_nargs(t));
    for (i=1; i<=tr_nargs(t); ++i) {
        printf (" %5d", tr_arg_int(t,i));
    }
    printf ("\n");
}

```

## watchdog

### Usage

```
./watchdog cpu_mask
```

This program illustrates how to monitor a certain condition in real-time and then act upon it accordingly.

In this case, the input to the program is the output of a NightTrace kernel daemon. The program watches for any context switches on the CPU specified in *cpu\_mask*.

For simplicity, this program only lists the time at which the context switch occurred and the process being switched in.

This program may be invoked with the following command:

```
ntracekd --stream /tmp/handle | ./watchdog 1
```

or it can be launched from the NightTrace GUI as part of a streaming kernel daemon definition (via the setting of the **Stream** checkbox on the **General** page of the **Daemon Definition** dialog (see “Stream” on page 5-48)).

See “Sample Programs” on page 3-107 for other programs demonstrating the capabilities of the NightTrace Analysis Application Programming Interface

## watchdog.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <ntrace_analysis.h>

// Example watchdog program; detect context switches on
// shielded CPU

// Usage: ./watchdog cpu_mask

// stdin is assumed to be the output of ntracekd (or watchdog
// was launched from the NightTrace GUI which set stdin to
// daemon output).

static void print (tr_t, tr_cond_t c, tr_offset_t, int, void *, int *);

int
main (int argc, char * argv[])
{
    tr_t t;
    tr_string_node_t * list;
    tr_offset_t offset;
```

```

tr_cond_t cond;
int i;
int cpu;
int errs;

if (argc != 2) {
    printf ("Usage: ntracekd --stream handle | watchdog cpu_mask\n");
    exit(1);
}
if (isatty(0)) {
    printf ("error: expect stdin to be streaming data from ntracekd\n");
    exit(1);
}
cpu = atoi(argv[1]);
if (cpu == 0) {
    printf ("error: cpu_mask must be a MASK of CPU bits\n");
    exit(1);
}

// Initialize the API
t = tr_init();

// Create a condition detecting context switches on specified CPU
// and register a callback for it.
cond = tr_cond_create(t,"switch");
tr_cond_id(t,cond,4150);
tr_cond_cpu(t,cond,cpu);
tr_cond_cb(t,cond,print,0);

// Open the input stream
tr_open_stream(t, 0, 1024*1024*50, 0);

// Ensure all is copasetic
errs = tr_error_check(t,&list);
if (errs) {
    for (i=0; i<errs; ++i)
        printf ("  %s (%s)\n", list[i].value, strerror(list[i].item));
    exit(1);
}

// Process all events
tr_iterate(t);

errs = tr_error_check(t,&list);
if (errs) {
    for (i=0; i<errs; ++i)
        printf ("  %s (%s)\n", list[i].value, strerror(list[i].item));
}

tr_close(t);
}

static
void
print (tr_t      t,
      tr_cond_t c,
      tr_offset_t offset,
      int        occurrence,
      void       * context,

```

```
        int          * disable)
{
    int pid = tr_pid(t);
    char * name = tr_process_name(t);

    if (!name) name = "<unknown>";

    printf ("context switch: %8.9f %5d %s\n", tr_time(t), pid, name);
}
```

## ptime

This program illustrates how to use the NightTrace GUI to export complex conditions and states to a source file which uses the API.

### Usage

```
./ptime kernel_trace_file
```

In this case, **ptime.c** contains the main program and the callback functions; we use the GUI to export an initialization routine which defines the states and registers the callbacks.

A NightTrace session file, **ptime.session**, is provided in this directory which contains a definition of a state called **ksoftirqd**.

In order to build the program **ptime**, you need to invoke NightTrace and export the state:

```
ksoftirqd
```

to generate the source file **export\_0.c**.

1. Issue the following command:

```
ntrace ptime.session
```

2. From the NightTrace menu, select the Export API Source File... menu item.
3. Select **ksoftirqd** in the list.
4. Clear checkbox for **Generate main() function**
5. Clear checkbox for **Generate callback function definitions**
6. Click on **Export Selected**
7. Click on **Close**
8. From the NightTrace menu, select **Exit Immediately**

### NOTE

Optionally, NightTrace can create a main program and callback bodies for you as well.

The **ksoftirqd** state tracks when the process **ksoftirqd/0** is active on CPU 0.

The **ptime** program simply collects the durations of each occurrence of the state and prints the total time at the end of the program.

To generate the *kernel\_trace\_file*, issue the following command:

```
ntracekd --wait=5 /tmp/kernel-data
```

You may then invoke the program:

```
./ptime /tmp/kernel-data
```

See “Sample Programs” on page 3-107 for other programs demonstrating the capabilities of the NightTrace Analysis Application Programming Interface

## ptime.c

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <ntrace_analysis.h>

// Example to calculate the amount of time the Kernel daemon
// ksoftirqd/0 spends processing on the CPU.

// The purpose of this example is to demonstrate use of the
// NightTrace GUI export feature to aid in forming conditions,
// states, and registering callbacks.

// Usage: ./ptime kernel_data_file

static double time = 0.0;

extern void tr_session_init(tr_t);

int
main (int argc, char * argv[])
{
    tr_t t;
    tr_string_node_t * list;
    tr_offset_t offset;
    tr_cond_t cond;
    int i;
    int errs;

    if (argc < 2) {
        printf ("Usage: search data_file\n");
        exit(1);
    }

    // Initialize the API and open the input data file
    t = tr_init();
    errs = tr_open_file(t, argv[1]);

    // Invoke the initialization function generated by the
    // NightTrace GUI to form string tables, conditions,
    // expressions, and register callbacks.
    if (!errs) {
        tr_session_init(t);
    }
}
```



```

        tr_activate(t);
    }

    // Ensure all is copasetic
    errs = tr_error_check(t,&list);
    if (errs) {
        for (i=0; i<errs; ++i)
            printf ("  %s (%s)\n", list[i].value, strerror(list[i].item));
        exit(1);
    }

    // Process all events
    tr_iterate(t);

    tr_close(t);
    tr_destroy(&t);

    printf ("ksoftirqd/0 used %9.8f seconds of CPU time\n", time);
}

void
ksoftirqd_start_func (tr_t input, tr_state_t state,
                     tr_offset_t offset, int occurrence,
                     void * context, int * disable) {
}

void
ksoftirqd_end_func (tr_t input, tr_state_t state,
                   tr_offset_t offset, int occurrence,
                   void * context, int * disable) {
    tr_state_info_t info;
    tr_state_info(input,state,&info);
    time += info.duration;
}

```

## browse

### Usage

```
./browse [-e expression] data_file
```

This program contains a collection of code segments which might be useful for reference.

It implements a simple command-line oriented browser.

### NOTE

The **browse** program is included mainly for reference; the NightTrace GUI is *much* more suitable for interactive browsing.

See “Sample Programs” on page 3-107 for other programs demonstrating the capabilities of the NightTrace Analysis Application Programming Interface

## browse.c

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include "ntrace_analysis.h"

// This test program implements a command-line orienter
// browser. It is provided because some of the code
// segments may be useful for reference. The NightTrace
// GUI tool is *much* more suitable for interactive browsing.


```

```

    if (process && process[0]) {
        printf ("%5d pid=%s %3d %8.9f %1d", offset, process, tr_id(t), time,
tr_nargs(t));
    } else {
        printf ("%5d pid=%d %3d %8.9f %1d", offset, tr_pid(t), tr_id(t), time,
tr_nargs(t));
    }
    for (i=1; i<=tr_nargs(t); ++i) {
        printf (" %5d", tr_arg_int(t,i));
    }
    printf ("\n");
}

static
void
print_event (tr_offset_t offset)
{
    int i;
    double time = tr_time_(t,offset);

    printf ("%5d %5d %3d %8.9f %1d", offset, tr_pid_(t,offset),
tr_id_(t,offset), time, tr_nargs_(t,offset));
    for (i=1; i<=tr_nargs_(t,offset); ++i) {
        printf (" %5d", tr_arg_int_(t,i,offset));
    }
    printf ("\n");
}

typedef enum { CMD_LIST,
CMD_NEXT,
CMD_PREV,
CMD_SEEK,
CMD_SEARCH,
CMD_COPY_FILE,
CMD_STATE,
CMD_CONDITION,
CMD_CALLBACK,
CMD_ITERATE,
CMD_REWIND,
CMD_QUIT,
CMD_UNKNOWN}

    commands;

static commands last_cmd = CMD_QUIT;

static int cond1 (tr_t t, tr_offset_t offset, void * v)
{
    return tr_nargs_(t,offset) > 0 && tr_arg_int_(t,1,offset) > 10;
}
static int cond2 (tr_t t, tr_offset_t offset, void * v)
{
    return tr_time_(t,offset) < 0.03712;
}
static int cond3 (tr_t t, tr_offset_t offset, void * v)
{
    return tr_nargs_(t,offset) > 0 && tr_arg_int_(t,1,offset) > 10;
}
static int cond4 (tr_t t, tr_offset_t offset, void * v)

```

```

{
    return tr_nargs_(t,offset) == 4;
}
static int cond5 (tr_t t, tr_offset_t offset, void * v)
{
    return tr_id_(t,offset) % 2 == 0;
}

static
void
event_cb (tr_t t, tr_cond_t c, tr_offset_t offset,
          int count, void * context, int * disable)
{
    printf ("event callback function\n");
    print(offset);
}

static
void
state_cb (tr_t t, tr_state_t s, tr_offset_t offset, int count, void * context,
          int * disable)
{
    tr_state_info_t info;
    print (offset);
    printf ("state callback function\n");
    tr_state_info (t, s, &info);
    printf ("    active      = %d\n", tr_state_active(t,s));
    printf ("    start_offset = %d\n", info.start_offset);
    printf ("    end_offset   = %d\n", info.end_offset);
    printf ("    gap          = %12.9fs\n", info.gap);
    printf ("    duration     = %12.9fs\n", info.duration);
}

static
commands
get_cmd (void)
{
    get_line(": ");

    if (strcmp(buffer,"") == 0) {
        return last_cmd;
    } else if (!strcmp(buffer,"list")) {
        return last_cmd=CMD_LIST;
    } else if (!strcmp(buffer,"next")) {
        return last_cmd=CMD_NEXT;
    } else if (!strcmp(buffer,"prev")) {
        return last_cmd=CMD_PREV;
    } else if (!strcmp(buffer,"seek")) {
        return last_cmd=CMD_SEEK;
    } else if (!strcmp(buffer,"search")) {
        return last_cmd=CMD_SEARCH;
    } else if (!strcmp(buffer,"copy_file")) {
        return last_cmd=CMD_COPY_FILE;
    } else if (!strcmp(buffer,"iterate")) {
        return last_cmd=CMD_ITERATE;
    } else if (!strcmp(buffer,"state")) {
        return last_cmd=CMD_STATE;
    } else if (!strcmp(buffer,"condition")) {

```

```

        return last_cmd=CMD_CONDITION;
    } else if (!strcmp(buffer,"callback")) {
        return last_cmd=CMD_CALLBACK;
    } else if (!strcmp(buffer,"rewind")) {
        return last_cmd=CMD_REWIND;
    } else if (!strcmp(buffer,"quit")) {
        return last_cmd=CMD_QUIT;
    } else {
        return last_cmd=CMD_UNKNOWN;
    }
}

static
void
do_search (void)
{
    tr_cond_t c;
    tr_dir_t dir;
    tr_offset_t o;

    get_line ("forward or backward (f/b): ");
    if (buffer[0] == 'b') {
        dir = tr_backward;
    } else {
        dir = tr_forward;
    }

    get_line ("enter name of condition to search for: ");
    c = tr_cond_find(t,buffer);
    if (c == TR_NO_COND) {
        printf ("could not locate condition \"%s\"\n", buffer);
        return;
    }
    o = tr_search (t, dir, c);
    if (o == TR_EOF) {
        printf ("Event Not Found\n");
    } else {
        print_event(o);
    }
}

static char * expression;

static
void
prime (void)
{
    tr_cond_t c1, c2, c3, c4, c5;
    char * err;

    c1 = tr_cond_create(t,"_cond1");
    tr_cond_func_and(t,c1,cond5,0);

    c2 = tr_cond_create(t,"_cond2");
    tr_cond_func_and(t,c2,cond4,0);

    c3 = tr_cond_create(t,"_cond3");
    tr_cond_id_range (t, c3, 50, 60);
}

```

```

c4 = tr_cond_create(t, "_test");
err = tr_cond_expr_and(t, c4, expression);
if (err) {
    printf ("%s\n", err);
}

c5 = tr_cond_create(t, "_cond5");
tr_cond_pid_name(t, c5, "foo");

tr_activate(t);

#if 0
{
    char * errs;
    int i;

    tr_error_clear(t);
    tr_session_init(t);
    errs = tr_error_check(t, &list);
    if (errs) {
        printf ("tr_session_init() failed:\n");
    }
    for (i=0; i<errs; ++i)
        printf ("  %s (%s)\n", list[i].value, strerror(list[i].item));
}
#endif
}

static
void
def_state (void)
{
    tr_state_t s;
    int error;
    int i;
    int low[2], high[2];
    tr_cond_t cond[2];

    for (i=0; i<2; ++i) {
        const char * prompt = (i ? "end: " : "start: ");
        write (1, prompt, strlen(prompt));
        get_line ("enter low bound of id range: ");
        low[i] = atoi(buffer);
        get_line ("enter high bound of id range: ");
        high[i] = atoi(buffer);
    }

    for (i=0; i<2; ++i) {
        const char * prompt = (i ? "end: " : "start: ");
        write (1, prompt, strlen(prompt));
        get_line ("enter condition name or <enter> for none: ");
        if (buffer[0] == '\0') {
            cond[i] = TR_NO_COND;
        } else {
            cond[i] = tr_cond_find(t, buffer);
            if (cond[i] == TR_NO_COND) {
                printf ("no such condition\n");
                return;
            }
        }
    }
}

```

```

    }
}

get_line ("enter name of state to be defined: ");

s = tr_state_create (t, buffer);
if (s == TR_NO_STATE) {
    printf ("state creation failed\n");
    return;
}

error = tr_state_start_id_range(t,s,low[0],high[0]);
error |= tr_state_end_id_range(t,s,low[1],high[1]);
if (cond[0] != TR_NO_COND) {
    tr_state_start_cond(t,s,cond[0]);
}
if (cond[1] != TR_NO_COND) {
    tr_state_end_cond(t,s,cond[1]);
}
if (error) {
    printf ("configuration of state failed\n");
    return;
}

tr_activate(t);

printf ("state \"%s\" has been successfully configured\n", buffer);
}

static
void
def_condition (void)
{
    tr_cond_t c;
    int low, high;
    int cpu;
    int pid;
    int error;
    int and_;
    tr_cond_func_t func;

    get_line ("enter low bound of id range or <enter> for none: ");
    low = atoi(buffer);
    get_line ("enter high bound of id range or <enter> for none: ");
    high = atoi(buffer);
    get_line ("enter cpu bias or <enter> for none: ");
    cpu = atoi(buffer);
    get_line ("enter pid or <enter> for none: ");
    pid = atoi(buffer);
    get_line ("enter name of condition to be defined: ");

    c = tr_cond_create (t, buffer);
    if (c == TR_NO_COND) {
        printf ("condition creation failed\n");
        return;
    }

    error = 0;

```

```

if (low) error |= tr_cond_id_range(t,c,low,high);
if (cpu)      tr_cond_cpu(t,c,cpu);
if (pid) error |= tr_cond_pid(t,c,pid);

for (;;) {
    get_line ("enter \"and\", \"or\", or <enter> for function conditions: ");
    if (buffer[0] == '\\0') break;
    else if (!strcmp(buffer,"and")) and_ = 1;
    else if (!strcmp(buffer,"or"))  and_ = 0;
    else {
        printf ("illegal response\n");
        return;
    }
    get_line ("enter condition callback function or expression: ");
    func = NULL;
    if (!strcmp(buffer,"cond1")) { func = cond1; }
    else if (!strcmp(buffer,"cond2")) { func = cond2; }
    else if (!strcmp(buffer,"cond3")) { func = cond3; }
    else if (!strcmp(buffer,"cond4")) { func = cond4; }
    else if (!strcmp(buffer,"cond5")) { func = cond5; }
    else func = NULL;
    if (func == NULL) {
        char * err;
        if (and_)
            err = tr_cond_expr_and(t,c,buffer);
        else
            err = tr_cond_expr_or(t,c,buffer);
        if (err) {
            printf ("invalid expression:\n%s\n",err);
            error = 1;
        }
    } else {
        if (and_) {
            error |= tr_cond_func_and(t,c,func,0);
        } else {
            error |= tr_cond_func_or(t,c,func,0);
        }
    }
}

if (error) {
    printf ("configuration of condition failed\n");
} else {
    printf ("condition has been successfully configured\n");
}

tr_activate(t);
}

static
void
destroy_callback (void)
{
    tr_cb_t id;

    get_line ("enter callback id to cancel: ");
    id = atoi(buffer);
    printf ("cancelling callback with ID %d\n", id);
    tr_cancel_cb (t, id);
}

```



```

}

static
void
def_callback (void)
{
    tr_cond_t c;
    tr_state_t s;
    int is_state;
    int id;
    tr_state_action_t a;

    get_line ("create or destroy a callback? (c/d) [c]: ");
    if (buffer[0] == 'd') {
        destroy_callback();
        return;
    }

    get_line ("state or condition callback? (s/c): [c]: ");
    is_state = buffer[0] == 's';

    if (is_state) {
        get_line ("enter state callback trigger: start, end, active, inactive: ");
        if (!strcmp(buffer,"start"))    a = tr_state_start_action;
        else if (!strcmp(buffer,"end"))  a = tr_state_end_action;
        else if (!strcmp(buffer,"active")) a = tr_state_active_action;
        else if (!strcmp(buffer,"inactive")) a = tr_state_inactive_action;
        else {
            printf ("illegal response\n");
            return;
        }
        get_line ("enter state name: ");
        s = tr_state_find(t,buffer);
        if (s == TR_NO_STATE) {
            printf ("unable to locate state \"%s\"\n", buffer);
            return;
        }
        id = tr_state_cb (t, s, a, state_cb, 0);
    } else {
        get_line ("enter condition name: ");
        c = tr_cond_find(t,buffer);
        if (c == TR_NO_COND) {
            printf ("unable to locate condition \"%s\"\n", buffer);
            return;
        }
        id = tr_cond_cb (t, c, event_cb, 0);
    }

    if (id == TR_NO_CB) {
        printf ("callback registration failed\n");
    } else {
        printf ("callback for %s \"%s\" was successfully registered as id %d\n",
            (is_state ? "state" : "condition"), buffer, id);
    }
}

int
main (int argc, char * argv[])
{

```

```

int status;
int i;
int done = 0;
int arg = 1;
int streaming = 0;
int cmd;
tr_offset_t o;
char buffer[100];

expression = "true";

for (;;) {
    if (argc < 2) {
        printf ("usage: %s [options] trace_data_file\n", argv[0]);
        printf ("options:\n"
            "    -e expr (expr)   Create an expression named \"_test\"\n"
            "                        using \"expr\" as the expression\n"
            "\n"
            "If \"trace_data_file\" is \"-\", then we assume stdin\n"
            "is a stream from a NightTrace daemon\n");
        exit(1);
    }
    if (argv[arg][0] == '-') {
        if (!strcmp(argv[arg], "-e")) {
            --argc;
            expression = argv[++arg];
        } else if (!strcmp(argv[arg], "-")) {
            streaming = 1;
            break;
        } else {
            argc = 0;
        }
    } else {
        break;
    }
    ++arg;
    --argc;
}

t = tr_init();

if (streaming) {
    input = fopen("/dev/tty", "r");
    //status = tr_open_stream(t, 0, 1024*1024*20, TR_STREAM_SAVE);
    status = 1;
} else {
    input = stdin;
    status = tr_open_file(t, argv[arg]);
}

if (status) {
    tr_string_node_t * list;
    int errs;
    printf ("tr_open_*() failed:\n");
    errs = tr_error_check(t, &list);
    for (i=0; i<errs; ++i)
        printf ("    %s (%s)\n", list[i].value, strerror(list[i].item));
    exit(1);
}

```

```

prime();

cmd = -1;

while (!done) {

    switch (cmd) {

    case CMD_LIST:
        for (;;) {
            o = tr_next_event(t);
            if (o == TR_EOF) break;
            print(o);
        }
        break;

    case CMD_NEXT:
        o = tr_next_event(t);
        print(o);
        break;

    case CMD_PREV:
        o = tr_prev_event(t);
        print(o);
        break;

    case CMD_SEEK:
        printf ("Input event offset of interest: ");
        fflush (stdout);
        o = atoi(fgets(&buffer[0],sizeof(buffer),input));
        printf ("seeking to %d\n", o);
        o = tr_seek(t,o);
        print(o);
        break;

    case CMD_SEARCH:
        do_search();
        break;

    case CMD_COPY_FILE:
        {
            tr_cond_t c;
            c = tr_cond_find(t, "copy");
            if (c == TR_NO_COND) {
                printf ("you must first define a condition called \"copy\"\n");
            } else {
                get_line ("Enter output file name: ");
                if (tr_copy_input(t,buffer,c,0666)) {
                    printf ("failed to write events\n");
                }
            }
        }
        break;

    case CMD_STATE:
        def_state();
        break;

    case CMD_CONDITION:

```

```
        def_condition();
        break;

    case CMD_CALLBACK:
        def_callback();
        break;

    case CMD_ITERATE:
        tr_iterate(t);
        break;

    case CMD_REWIND:
        (void) tr_seek(t,-1);
        break;

    case CMD_QUIT:
        done = 1;
        continue;
        //break;

    default:
        printf ("Commands:\n"
            "  list\n"
            "  next\n"
            "  prev\n"
            "  seek\n"
            "  search\n"
            "  copy_file\n"
            "  state\n"
            "  condition\n"
            "  callback\n"
            "  iterate\n"
            "  rewind\n"
            "  quit\n");
    }

    cmd = get_cmd();

} while (!done);

tr_close (t);
tr_destroy (&t);

return 0;
}
```

## Invoking NightTrace

---

Command-line Options .....	4-1
Summary Criteria .....	4-5
Command-line Arguments .....	4-9
Trace Event Files .....	4-10
Event Map Files .....	4-10
Page Configuration Files .....	4-13
Tables .....	4-13
String Tables .....	4-15
Pre-Defined String s .....	4-16
Format Tables .....	4-19
Pre-Defined Format Tables .....	4-23
Session Configuration Files .....	4-24
Trace Data Segments .....	4-25



## Invoking NightTrace

NightTrace is invoked using **ntrace** which is normally installed in `/usr/bin`.

The full command syntax for **ntrace** is:

```
ntrace    [-h] [--help] [--help-summary]
          [-v] [--version] [-l] [--listing]
          [--stats] [-n] [--notimer]
          [-s val] [--start={ offset | time { s | u } | percent% }]
          [-e val] [--end={ offset | time { s | u } | percent% }]
          [-hm] [--hide-main-window] [-Xoption ...]
          [-u] [--use-session] [--summary=criteria]
          [--verbose]
          [file ...]
```

Depending on the options and arguments specified to **ntrace**, NightTrace:

- loads *all* trace event information into memory
- checks the syntax of specifications in each file argument
- processes each file argument
- loads any display pages and their objects into memory
- presents any display pages (see Chapter 9 “Display Pages”)
- displays the NightTrace Main Window (see Chapter 5 “Using the Night-Trace GUI”)

## Command-line Options

The command-line options to **ntrace** are:

**-h**  
**--help**

Display **ntrace** invocation syntax and a list of all command line options to standard output.

**--help-summary**

Display help specific to the **--summary** option to standard output.

See “Summary Criteria” on page 4-5 for more information.

**-v**

**--version**

Display the current version of NightTrace to standard output and exit.

**-l**

**--listing**

Display a chronological listing of all trace events and their arguments from all supplied trace-event data files to standard output and exit.

The output includes the following information about a trace event:

- relative timestamp
- trace event ID
- any trace event argument(s)
- the global process identifier (PID), process name, or thread name
- the CPU

The timestamp for the first trace event is zero seconds (0s). All other timestamps are relative to the first one.

If you supply an event map file on the invocation line, NightTrace displays symbolic trace event names instead of numeric trace event IDs, and displays trace event arguments in the format you specify in the file, rather than the hexadecimal default format. For more information on event map files, see “Event Map Files” on page 4-10.

#### **NOTE**

The CPU field is only meaningful for kernel trace events; for user trace events, the CPU field is displayed as CPU=??.

**--stats**

Display simple overall statistics about the trace-event data files to standard output and exit.

The statistics are grouped by trace event file, with cumulative statistics for all trace event files.

The statistics include:

- the number of trace event files
- their names
- the number of trace events logged
- the number of trace events lost



For example, the following command:

```
ntracekd --wait=2 kernel-data
```

collects kernel trace data for two seconds from the system on which it was issued and saves the results to **kernel-data** (see Chapter 7 “Generating Trace Event Logs with ntracekd”).

Issuing the command:

```
ntrace --stats kernel-data
```

results in the output similar to the following:

```
Read 1 trace event segment timestamped with Intel TSC.
(1) Kernel trace event log file: kernel-data.
    226809 trace events plus 204596 continuation events.
    105419 trace events lost.
    2.9707482s time span, from 0.0000000s to 2.9707482s.

    226809 total events read from disk plus 204596 continuation events.
    226808 total events saved in memory; 117 events internal to ntrace.
    105419 total trace events lost.
    2.9707482s total time span saved in memory.
```

Detailed summary information about a trace data set is available via the **--summary** option (see page 4-4).

```
-n  
--notimer
```

Exclude from analysis trace events for system timer interrupts in the kernel trace file.

```
-s val  
--start={ offset | time{ s | u } | percent% }
```

Exclude from analysis trace events before the specified trace-event offset, relative time in seconds (**s**) or microseconds (**u**), or percent of total trace events.

The specified values can be:

<i>offset</i>	Load trace events after the specified trace event offset. (See “Grid” on page 9-28 for information about trace event offsets.)
<i>time{ s   u }</i>	Load trace events after the specified relative time in seconds ( <b>s</b> ) or microseconds ( <b>u</b> ).
<i>percent%</i>	Load trace events after the specified percent of total trace events. The % is required.

If you invoke NightTrace with several **--start** options, NightTrace pays attention only to the last one.

**-e** *val*  
**--end=**{ *offset* | *time*{ **s** | **u** } | *percent%* }

Exclude from analysis trace events after the specified trace-event offset, relative time in seconds (**s**) or microseconds (**u**), or percent of total trace events.

The specified values can be:

<i>offset</i>	Load trace events before the specified trace event offset.
<i>time</i> { <b>s</b>   <b>u</b> }	Load trace events before the specified relative time in seconds ( <b>s</b> ) or microseconds ( <b>u</b> ).
<i>percent%</i>	Load trace events before the specified percent of total trace events. The % is required.

If you invoke NightTrace with several **--end** options, NightTrace pays attention only to the last one.

**-hm**  
**--hide-main-window**

Start NightTrace with the NightTrace Main Window hidden (see Chapter 5 “Using the NightTrace GUI”); only display pages are shown (see Chapter 9 “Display Pages”).

The NightTrace Main Window may be subsequently displayed using the NightTrace Main Window... menu item on the Page menu from any display page (see “Page” on page 9-3).

**-u**  
**--use-session**

Automatically load the last session used in a previous invocation of NightTrace. All files associated with the previous session are automatically loaded.

**--summary=***criteria*

Provide a textual summary of specified trace events using the supplied *criteria*. Summary results are sent to standard output.

See “Summary Criteria” on page 4-5 for details regarding valid *criteria*.

**--verbose**

In addition to the cumulative statistics normally output, this option provides detailed information about each occurrence of the item being summarized.

**-Xoption ...**

Use any standard X Toolkit command line options. See **X(1)**.

*file ...*

You can invoke NightTrace with arguments such as trace event files, event map files, page configuration files, session configuration files, or trace data segments.

See “Command-line Arguments” on page 4-9 for a description of these types of files.

By default, when NightTrace starts up, it reads and loads *all* trace events from all trace event files into memory. The `--process`, `--start`, and `--end` options let you prevent the loading (but not the reading) of certain trace events.

For example, the following invocation displays only those trace events logged 0.5 seconds or more after the start of the data set.

```
ntrace --start=0.5s kernel-data
```

## Summary Criteria

The `--summary` option is supplied with criteria for command-line usage without ever using the GUI to perform summaries.

### NOTE

The `--verbose` option (see “`--verbose`” on page 4-4) provides detailed information about each occurrence of the item being summarized in addition to the cumulative statistics normally output.

This criteria consists of a comma-separated list of any of the following:

*crit*

This allows previously-defined summary criteria to be referenced when doing command line summaries.

Summary criteria may be named with a criteria tag (see “Criteria Tag” on page 12-21) in the **Summarize NightTrace Events** dialog (see “Summarizing Statistical Information” on page 12-12). The tagged criteria can then be saved into a NightTrace session configuration file (see “Session Configuration Files” on page 4-24).

To use previously-defined summary criteria when executing a summary from the command line, specify the desired criteria tag (*crit*) on the command line along with the NightTrace session configuration file which contains that tag.

**ev**: *event*

Summarize the number of occurrences of the specified *event*.

**p**: *process*

Summarize all events associated with the specified *process*.

**t:thread**

Summarize all events associated with the specified *thread*.

**qe:name**

Summarize all occurrences of the qualified event *name*.

**qs:name**

Summarize all occurrences of the qualified state *name*.

**s:call**

Summarize all events associated with the entry or resumption of the specified system *call*.

**s1:call**

Summarize all events associated with the exit or suspension of the specified system *call*.

**se:call**

Summarize all events associated with the specified system *call*.

**ss:call**

Summarize all occurrences of a state defined by system call activity for the specified system *call*.

**i:intr**

Summarize all events associated with the entry or resumption of the specified interrupt *intr*.

**i1:intr**

Summarize all events associated with the exit or interruption of the specified interrupt *intr*.

**ie:intr**

Summarize all events associated with the specified interrupt *intr*.

**is:intr**

Summarize all occurrences of a state defined by interrupt activity for the specified interrupt *intr*.

**e:exc**

Summarize all events associated with the entry or resumption of the specified exception *exc*.

**e1:exc**

Summarize all events associated with the exit or interruption of the specified exception *exc*.

**ee:exc**

Summarize all events associated with the specified exception *exc*.

**es:exc**

Summarize all occurrences of a state defined by exception activity for the specified exception *exc*.

**skip:on**

Suppresses summarization for all subsequent criteria in the list (or until a **skip:off** criteria is seen) if there are no summarization matches for the criteria.

**skip:off**

Reactivates summarization for all subsequent criteria in the list (or until a **skip:on** criteria is seen) if there are no summarization matches for the criteria.

**st:start-end**

Summarize all occurrences of the state defined by the starting event *start* and terminated by the ending event *end*.

These may be combined together along with tagged criteria from the **Summarize NightTrace Events** dialog (see “Summarizing Statistical Information” on page 12-12) in a comma-separated list.

Consider the following example:

```
ntrace --summary=ss:read,ss:alarm,ev:5,crit_0 event_file my_session
```

Using the trace event file **event\_file** as the trace data source (see “Trace Event Files” on page 4-10), NightTrace will:

1. summarize the number of occurrences of `read` and `alarm` system call states that occur in the data source; provide information pertaining to the duration of each state (`min`, `max`, `avg`, `sum`); and provide information related to the gaps between each state (`min`, `max`, `avg`, `sum`)
2. summarize the number of occurrences of user events with a *trace event ID* of 5 as well as information about the gaps between the events (`min`, `max`, `avg`)
3. perform a summary using the criteria defined by criteria tag `crit_0` (see “Criteria Tag” on page 12-21) in the **my\_session** session file (see “Session Configuration Files” on page 4-24)

## NOTE

In order to use a summary criteria tag on the command line, the NightTrace session configuration file in which it was defined must be specified on the command line as well (see “Session Configuration Files” on page 4-24).

The following criteria may be specified *alone* (not part of a comma-separated list):

**k**[ :*proc*]

Summarize kernel states: system calls, exceptions, and interrupts. If :*proc* is provided, only those states involving process *proc* are summarized.

**ksc**[ :*proc*]

Summarize kernel system call durations. If :*proc* is provided, only those system calls involving process *proc* are summarized.

**kexc**[ :*proc*]

Summarize kernel exception durations. If :*proc* is provided, only those exceptions involving process *proc* are summarized.

**kintr**[ :*proc*]

Summarize kernel interrupt durations. If :*proc* is provided, only those interrupts involving process *proc* are summarized.

**evt**[ :*proc*]

Summarize the number of occurrences of all events named in event map files. User events which are not named in event map files are not shown. If :*proc* is provided, only those events associated with *proc* are summarized.

*proc*

Summarize the number of events for each process.

## Command-line Arguments

You can supply filenames as arguments to the **ntrace** command when invoking Night-Trace. These files may contain trace event data, display page layouts, additional configuration information, or information related to a previously-saved session.

These arguments can be:

- trace event files

Trace event files are captured by a user or kernel trace daemon and contain sequences of trace events logged by your application or the operating system kernel.

See “Trace Event Files” on page 4-10 for more information.

- event map files

Event map files map short mnemonic trace event names to numeric trace event IDs and associate data types with trace event arguments. These ASCII files are created by the user.

See “Event Map Files” on page 4-10 for more information.

- page configuration files

Configuration files define display pages, the display objects contained within them, string tables, and format tables. These ASCII files are usually created by Night-Trace.

See “Page Configuration Files” on page 4-13 for more information.

- session configuration files

Session configuration files define a list of daemon sessions and their individual configurations. In addition, session configuration files contain definitions of macros, qualified events, qualified states, and search and summary configurations from previous uses of the session. Also, session configuration files contain a list of any files the user associated with the session, such as event map files and trace data files.

See “Session Configuration Files” on page 4-24 for more information.

- trace data segments

Trace data segments are conglomerations of all trace data saved in a much more efficient format than raw trace event files providing for faster initialization at startup. These files are created using the **Save Data Segment...** button on the Night-Trace Main Window (see “Save Data Segment...” on page 5-40).

See “Trace Data Segments” on page 4-25 for more information.

## Trace Event Files

Trace event files are created by user and kernel trace daemons. They consist of header information and individual trace events and their arguments as logged by user applications or the operating system. NightTrace detects trace event files as specified on the command line and does the required initialization processing so that the trace events contained in the files are available for display.

To load a trace event file, either:

- specify the trace event file as an argument to the **ntrace** command when you invoke NightTrace
- click on the **Open...** button in the Session Overview Area of the NightTrace Main Window (see “Session Overview Area” on page 5-39) and select the trace event file from the file selection dialog

## Event Map Files

NightTrace does not require you to use event map files. However, if you use these file(s), you can improve the readability of your NightTrace displays.

An *event map file* allows you to associate meaningful names with the more cryptic trace event ID numbers. It also allows you to associate additional information with a trace event including the number of arguments and the argument conversion specifications or display formats. Although NightTrace does not require you to use event map files, labels and display formats can make graphical NightTrace displays and textual summary information much more readable.

To load an existing event map file, perform any of the following:

- specify the event map file as an argument to the **ntrace** command when you invoke NightTrace
- click on the **Open...** button in the Session Overview Area of the NightTrace Main Window (see “Session Overview Area” on page 5-39) and select the event map file from the file selection dialog
- select the **Open Event Map File...** menu item from the **NightTrace** menu on the NightTrace Main Window (see “Open Event Map File...” on page 5-6)

You can create an event map file with a text editor before you invoke NightTrace. You may also select the **New Event Map File...** menu item from the **NightTrace** menu on the NightTrace Main Window (see “New Event Map File...” on page 5-6) which launches the editor defined by the `EDITOR` environment variable with a file initially containing a template describing the format of an event map file. The user may then populate the new event map file with associations of meaningful names with specific trace event IDs.

There is one trace event name mapping per line. White space separates each field except the conversion specifications; commas separate the conversion specifications. NightTrace ignores blank lines and treats text following a # as comments.



The syntax for the trace event mappings in the event map file follows:

```
event: ID "event_name" [ nargs [ conv_spec, ... ] ]
```

Fields in this file are:

*event*:

The keyword that begins all trace event name mappings.

*ID*

A valid integer in the range reserved for user trace events (0-4095, inclusive). Each time you call a NightTrace trace event logging routine, you must supply a trace event ID.

*event\_name*

A character string to be associated with *event\_ID*. Trace event names must begin with a letter and consist solely of alphanumeric characters and under-scores. Keep trace event names short; otherwise, NightTrace may be unable to display them in the limited window space available.

The following words are reserved in NightTrace and should not be used in uppercase or lowercase as trace event names:

- NONE
- ALL
- ALLUSER
- ALLKERNEL
- TRUE
- FALSE
- CALC

### TIP

Consider giving your trace events uppercase names in event map files and giving any corresponding qualified events the same name in lowercase. For more information about qualified events, see "Qualified Events" on page 11-113.

If your application logs a trace event with one or more numeric arguments, by default NightTrace displays these arguments in decimal integer format. To override this default, provide a count of argument values and one argument conversion specification or display format per argument.

*nargs*

The number of arguments associated with a particular trace event. If *nargs* is too small and you invoke NightTrace with the event map file and the

**--listing** option, NightTrace shows only *nargs* arguments for the trace event.

*conv\_spec*

A conversion specification or display format for a trace event argument. NightTrace uses conversion specification(s) to display the trace event's argument(s) in the designated format(s). There must be one conversion specification per argument. Valid conversion specifications for displays include the following:

<code>%d</code>	signed decimal integer (default)
<code>%o</code>	unsigned octal integer
<code>%x</code>	unsigned hexadecimal integer
<code>%lf</code>	signed double precision, decimal floating point

For more information on these conversion specifications, see **printf(3S)**.

The following line is an example of an entry in an event map file:

```
event: 5 "Error" 2 %x %lf
```

NightTrace displays trace event 5 and labels the trace event "Error". Trace event 5 also has two (2) arguments. NightTrace displays the first argument in unsigned hexadecimal integer (`%x`) format and the second argument in signed double precision decimal floating point (`%lf`) format. (You may override these conversion specifications when you configure display objects.)

For more information on event map files, see "Pre-Defined String s" on page 4-16 and the **ntrace(4)** man page.

## Page Configuration Files

A *page configuration file* contains information related to the layout of a particular display page and includes the configurations of all display objects that have been created on that page. In addition, any user-defined tables that have been created for that page is also contained in this file. Although NightTrace does not require you to use page configuration files, using a page configuration file improves the readability of your display pages and saves you time laying out your display pages.

A page configuration file is an ASCII file containing such definitions as:

- display page definitions (see Chapter 9 “Display Pages”)
- string table definitions (see “String Tables” on page 4-15)
- format table definitions (see “Format Tables” on page 4-19)

### NOTE

Any tables found in page configuration files are imported into the session; when the session is saved, these tables are saved with the session. Tables are no longer saved as part of the page configuration files.

### NOTE

If you define a string table or format table more than once in a configuration file, NightTrace merges the two tables; if there are duplicate entries, values come from the last definition.

You can create, modify, save, and load configuration files from within NightTrace; however, you must use a text editor to create and modify tables in a configuration file. NightTrace ignores blank lines and treats text between a `/*` and a `*/` as comments in configuration files; however, saving a configuration file removes your comments.

To load an existing configuration file, either:

- specify the configuration file as an argument to the `ntrace` command when you invoke NightTrace
- click on the **Open...** button in the Session Overview Area of the NightTrace Main Window (see “Session Overview Area” on page 5-39) and select the configuration file from the file selection dialog

## Tables

The page configuration file (see “Page Configuration Files” on page 4-13) may contain two types of tables, both of which can improve the readability of your NightTrace displays:

- string tables (see “String Tables” on page 4-15)

- format tables (see “Format Tables” on page 4-19)

A table lets you associate meaningful character strings with integer values such as trace event arguments. These character strings may appear in NightTrace displays.

The following table names are reserved in NightTrace and should not be redefined in uppercase or lowercase:

- event
- pid
- tid
- boolean
- name\_pid
- name\_tid
- node\_name
- pid\_nodename
- tid\_nodename
- vector
- syscall
- device
- vector\_nodename
- syscall\_nodename
- device\_nodename
- event\_summary
- event\_arg\_summary
- event\_arg\_dbl\_summary
- state\_summary

The results are undefined if you supply your own version of these tables.

#### **NOTE**

The only way to put tables into your configuration file is by text editing the file before you invoke NightTrace. To avoid any forward-reference problems, define all string tables before any format tables.

For more information on pre-defined tables, see “Pre-Defined Strings” on page 4-16, “Pre-Defined Format Tables” on page 4-23, and page 13-13.

If you define a string table or format table more than once in a configuration file, NightTrace merges the two tables; if there are duplicate entries, values come from the last definition.

## String Tables

You can log a trace event with one or more numeric arguments. Sometimes these arguments can take on a nearly fixed set of values. A *string table* associates an integer value with a character string. Labeling numeric values with text can make the values easier to interpret.

The syntax for a string table is:

```
string_table ( table_name ) = {
    item = int_const, "str_const" ;
    ...
    [ default_item = "str_const" ; ]
};
```

Include all special characters from the syntax except the ellipsis ( . . . ) and square brackets ([ ]).

The fields in a string table definition are:

*string\_table*

The keyword that starts the definition of all string tables.

*table\_name*

The unique, user-defined name of this table. This name describes the relationship of the numeric values in this string table.

An *item line* associates an integer value with a character string. This line extends from the keyword *item* through the ending semicolon. You may define any number of item lines in a single string table. The fields in an item line are:

*item*

The keyword that begins all item lines.

*int\_const*

An integer constant that is unique within *table\_name*. It may be decimal, octal, or hexadecimal. Decimal values have no special prefix. Octal values begin with a zero (0). Hexadecimal values begin with 0x.

*str\_const*

A character string to be associated with *int\_const*. Keep this string short; otherwise, NightTrace may be unable to display it in the limited window space available. Use a \n for a newline, not a carriage return in the middle of the string.

The optional *default item line* associates all other integer values (those not explicitly referenced) with a single string.

### TIP

If your table needs only one entry, you may omit the item line and supply only the default item line. A `get_string()` call with this table name as the first parameter needs no second parameter.

NightTrace returns a string of the item number in decimal if:

- there is no default item line, and the specified item is not found
- the string table is not found (The first time NightTrace cannot find a particular string table, NightTrace flags it as an error.)

The following lines provide an example of a string table in a configuration file.

```
string_table (curr_state) = {  
    item = 3, "Processing Data";  
    item = 1, "Initializing";  
    item = 99, "Terminating";  
    default_item = "Other";  
};
```

In this example, your application logs a trace event with a numeric argument that identifies the current state (`curr_state`). This argument has three significant values (3, 1, and 99). When `curr_state` has the value 3, the NightTrace display shows the string "Processing Data." When it has the value 1, the display shows "Initializing." When it has the value 99, the display shows "Terminating." For all other numeric values, the display shows "Other."

For more information on string tables and the `get_string()` function, see page 11-104.

## Pre-Defined String s

The following string tables are pre-defined in NightTrace:

`event`

The `event` string table is a dynamically generated table which contains all trace event names.

This table is indexed by an event code or an event code name. Examples of using this table are:

```
get_string(event, 4112)  
get_item(event, "TR_INTERRUPT_EXIT")
```

`pid`

A dynamically generated string table internal to NightTrace. In user tracing, it associates global process ID numbers with process names of the processes being traced. In kernel tracing, it associates process ID numbers with all active process names and resides in the dynamically generated **vectors** file.

**NOTE**

When analyzing trace event files from multiple systems, process identifiers are not guaranteed to be unique across nodes. Therefore, accessing the `pid` table may result in an incorrect process name being returned for a particular process ID. To get the correct process name for a process ID, the `pid` table for the node on which the process identifier occurs should be used instead. The `pid` table is maintained for backwards compatibility.

This table is indexed by a process identifier or a process name. Examples of using this table are:

```
get_string(pid, pid())
get_item(pid, "ntraceud")
```

See “Processes” on page 10-51 for more information.

`tid`

A dynamically generated string table internal to NightTrace. In user tracing, it associates NightTrace thread ID numbers with thread names. In kernel tracing, this table is not used.

**NOTE**

When analyzing trace event files from multiple systems, thread identifiers are not guaranteed to be unique across nodes. Therefore, accessing the `tid` table may result in an incorrect thread name being returned for a particular thread ID. To get the correct thread name for a thread ID, the `tid` table for the node on which the process identifier occurs should be used instead. The `tid` table is maintained for backwards compatibility.

This table is indexed by a thread identifier or a thread name. Examples of using this table are:

```
get_string(tid, tid())
get_item(tid, "cleanup_thread")
```

See “Threads” on page 10-52 for more information.

`boolean`

A string table which associates 0 with `false` and all other values with `true`.

`name_pid`

A dynamically generated string table internal to NightTrace. It maps all known node ID numbers (which are internally assigned by NightTrace) to the name of the node’s process ID table).

This table is indexed by a node identifier or a node name. Examples of using this table are:

```
get_string(name_pid, node_id())
get_item(name_pid, "system123")
```

Consider the following example:

```
get_string(get_string(name_pid,node_id()),pid)
```

The nested call to `get_string(name_pid,node_id())` returns the name of the process ID table on the system where this trace point was logged. We then index that table with the current process ID (since processes IDs are guaranteed to be unique when analyzing multiple trace event files obtained from multiple systems) to obtain the name of the current process.

#### NOTE

The predefined `process_name()` function is equivalent to the expression above - and much simpler to write! (See "process\_name()" on page 11-32 for more information.)

#### name\_tid

A dynamically generated string table internal to NightTrace. It maps all known node ID numbers (which are internally assigned by NightTrace) to the name of the node's thread ID table).

This table is indexed by a node identifier or a node name. Examples of using this table are:

```
get_string(name_tid, 1)
get_item(name_tid, "charon")
```

#### node\_name

A dynamically generated string table internal to NightTrace. It associates node ID numbers (which are internally assigned by NightTrace) with node names.

This table is indexed by a node identifier or a node name. Examples of using this table are:

```
get_string(node_name, node_id())
get_item(node_name, "gandalf")
```

#### pid\_nodename

A dynamically generated string table internal to NightTrace. In kernel tracing, it associates process ID numbers with all active process names for a particular node and resides in that node's **vectors** file. In user tracing, it associates global process ID numbers with process names of the processes being traced for a particular node.

This table is indexed by a process identifier or a process name. Examples of using this table are:



```
get_string(pid_sbcl, pid())
get_item(pid_engsim, "nfsd")
```

*tid\_nodename*

A dynamically generated string table internal to NightTrace. In kernel tracing, this table is not used. In user tracing, it associates NightTrace thread ID numbers with thread names for a particular node.

This table is indexed by a thread identifier or a thread name. Examples of using this table are:

```
get_string(tid_harpo, 1234567)
get_item(tid_shark, "reaper_thread")
```

## vector

See page 13-13.

## syscall

See page 13-13.

## device

See page 13-13.

## vector\_nodename

See page 13-13.

## syscall\_nodename

See page 13-13.

## device\_nodename

See page 13-13.

You can use pre-defined string tables anywhere that string tables are appropriate. Use the `get_string()` function to look up values in string tables. For information about the `get_string()` function, see page 11-104.

## Format Tables

Like string tables, *format tables* let you associate an integer value with a character string; however, in contrast to a string table string, a format table string may be dynamically formatted and generated. Labeling numeric values with text can make the values easier to interpret.

The syntax for a format table is:

```
format_table ( table_name ) = {
    item = int_const, "format_string" [ , "value1" ] ... ;
    ...
    [ default_item = "format_string" [ , "value1" ] ... ; ]
};
```

Include all special characters from the syntax except the ellipses (. . .) and square brackets ([]).

The fields in a format table are:

`format_table`

The keyword that begins the definition of all format tables.

*table\_name*

The unique, user-defined name of this table. This name describes the relationship of the numeric values in this format table.

An *item line* associates a single integer value with a character string. This line extends from the keyword `item` through the ending semicolon. You may have any number of item lines in a single format table.

The fields in an item line are:

`item`

The keyword that begins all item lines.

*int\_const*

An integer constant that is unique within *table\_name*. This value may be decimal, octal, or hexadecimal. Decimal values have no special prefix. Octal values begin with a zero (0). Hexadecimal values begin with 0x.

*format\_string*

A character string to be associated with *int\_const*. Keep this string short; otherwise, NightTrace may be unable to display it in the limited window space available. Use a `\n` for a newline, not a carriage return in the middle of the string.

The string contains zero or more conversion specifications or display formats. Valid conversion specifications for displays include the following:

<code>%i</code>	Signed integer
<code>%u</code>	Unsigned decimal integer
<code>%d</code>	Signed decimal integer
<code>%o</code>	Unsigned octal integer
<code>%x</code>	Unsigned hexadecimal integer
<code>%lf</code>	Signed double precision, decimal floating point
<code>%e</code>	Signed decimal floating point, exponential notation
<code>%c</code>	Single character
<code>%s</code>	Character string
<code>%%</code>	Percent sign
<code>\n</code>	Newline

For more information on these conversion specifications, see `printf(3S)`.

*format\_string* may contain any number of conversion specifications. There is a one-to-one correspondence between conversion specifications and quoted values. A particular conversion specification-quoted value pair must match in both data type and position. For example, if *format\_string* contains a `%s` and a `%d`, the first quoted value must be of type string and the second one must be of type integer. If the number or data type of the quoted value(s) do not match *format\_string*, the results are not defined.

#### *value1*

A value associated with the first conversion specification in *format\_string*. The value may be a constant string (literal) expression or a NightTrace expression. A string literal expression must be enclosed in double quotes. An expression may be a `get_string()` call (see page 11-104). For more information on expressions, see Chapter 11 “Using Expressions”.

The optional `default_item` line associates all other integer values with a single format item. NightTrace flags it as an error if an expression evaluates to a value that is not on an item line and you omit the default item line.

#### TIP

If your table needs only one entry, you may omit the item line and supply only the default item line. A `get_format()` call with this table name as the first parameter needs no second parameter.

The following lines provide an example of a string table and format table in a configuration file.

```
string_table (curr_state) = {
    item = 3, "Processing Data";
    item = 1, "Initializing";
    item = 99, "Terminating";
    default_item = "Other";
};

format_table (event_info) = {
    item = 186, "Search for the next time we process data";
    item = 25, "The current state is %s",
        "get_string (curr_state, arg1())";
    item = 999, "Current state is %s, current trace event is %d",
        "get_string (curr_state, arg1())",
        "offset()";
    default_item = "Other";
};
```

In this example, the first numeric argument associated with a trace event represents the current state (`curr_state`), and the `event_info` format table represents information associated with the trace event IDs. When trace event 186 occurs, a `get_format(event_info, 186)` makes NightTrace display:

```
Search for the next time we process data
```

When trace event 25 occurs, NightTrace replaces the conversion specification (`%s`) with the result of the `get_string()` call. If `arg1()` has the value 1, then NightTrace displays:

```
The current state is Initializing
```

When trace event 999 occurs, NightTrace replaces the first conversion specification (`%s`) with the result of the `get_string()` call and replaces the second conversion specification (`%d`) with the integer result of the numeric expression `offset()`. If `arg(1)` has the value 99 and `offset()` has the value 10, then NightTrace displays:

```
Current state is Terminating, current trace event is
10
```

For all other trace events, NightTrace displays "Other".

For more information on `get_string()`, see "get\_string()" on page 11-104.

For more information on format tables and the `get_format()` function, see "get\_format()" on page 11-108.

For more information about `arg1()`, see "arg()" on page 11-16.

For more information about `offset()`, see "offset()" on page 11-26.

## **Pre-Defined Format Tables**

The following format tables are pre-defined:

`state_summary`

Formats statistics about the state matches summarized, state durations, and state time gaps. This table provides the default state summary output format.

`event_summary`

Formats statistics about the trace event matches and trace event time gaps. This table provides the default trace event summary output format.

`event_arg_summary`

Formats statistics about the trace event matches and their type long trace event arguments.

`event_arg_dbl_summary`

Formats statistics about the trace event matches and their type double trace event arguments.

For more information about summaries, see “Summarizing Statistical Information” on page 12-12.

You can use pre-defined format tables anywhere that format tables are appropriate. Use the `get_format()` function to look up values in format tables. For information about the `get_format()` function, see “`get_format()`” on page 11-108.

## Session Configuration Files

A session configuration file defines a NightTrace session.

### NOTE

NightTrace remembers the last session loaded or saved on a per-user basis. To simplify restarting NightTrace at another time to analyze the same data, the usage of the **--use-session (-u)** command line option (see “-u --use-session” on page 4-4) is strongly encouraged to invoke NightTrace with the last session loaded or saved.

A session configuration may include:

- daemon definitions  
See “Daemon Definition Dialog” on page 5-41 for more information.
- display page configurations  
See “Page Configuration Files” on page 4-13 for more information.
- string tables
  - event names specified for user event IDs
  - any user-defined string tables
  - string tables imported from generated Ada display page configuration files
  - any modifications to default NightTrace string tables, or string tables embedded in trace data files
- qualified states, qualified events, and macro definitions  
See “Using Expressions” on page 11-1 for more information.
- named tags  
See “Tag” on page 9-32 for more information.
- previously-executed searches  
See “Searching for Points of Interest” on page 12-1 for more information.
- previously-executed summaries  
See “Summarizing Statistical Information” on page 12-12 for more information.
- references to saved trace data segment files  
See “Trace Data Segments” on page 4-25 for more information.

- references to kernel trace files generated by **ntracekd** (see “The ntracekd Daemon” on page 7-1), or a kernel daemon defined in the GUI (see “Kernel” on page 5-47)
- references to user trace files generated by **ntraceud** (see “The ntraceud Daemon” on page 6-1), or a user daemon defined in the GUI (see “User Application” on page 5-47)

Session configuration files can be generated by the following menu items in the **Night-Trace** menu of the NightTrace Main Window:

- **Save Session** (see “Save Session” on page 5-4)
- **Save Session Copy** (see “Save Session Copy” on page 5-5)
- **Save Session As...** (see “Save Session As...” on page 5-5)

Upon exiting when there are unsaved changes to the session, the user is given the chance to **Save Session and Exit** or **Save Session Copy and Exit**. See “Unsaved Changes” on page 5-7.

The user may load the session on a subsequent invocation of NightTrace by either:

- specifying the session configuration filename on the command-line when invoking **ntrace** (see “Invoking NightTrace” on page 4-1)
- using the **Open Session** dialog (see “Open Session...” on page 5-4) to open the session configuration file from the NightTrace Main Window

## Trace Data Segments

Trace data segments are conglomerations of all trace data saved in a much more efficient format than raw trace event files providing for faster initialization at startup.

Trace data segments are saved using the **Save Data Segment...** button on the Night-Trace Main Window (see “Save Data Segment...” on page 5-40).





## Using the NightTrace GUI

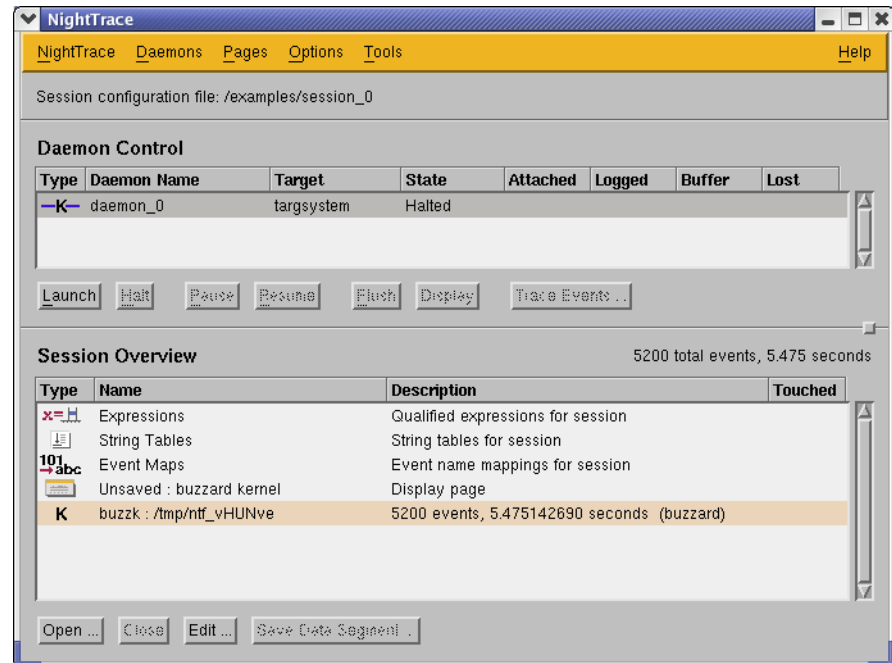
NightTrace Main Window Menu Bar .....	5-3
NightTrace .....	5-3
Unsaved Changes .....	5-7
Daemons .....	5-10
Login .....	5-14
Enter Password .....	5-14
Attach Daemons .....	5-16
Pages .....	5-18
Build Custom Kernel Page .....	5-21
Select Graphs .....	5-23
Options .....	5-25
Refresh Interval .....	5-26
Display Buffer Size Warning .....	5-27
Tools .....	5-28
Help .....	5-30
Session Configuration File Name Area .....	5-31
Daemon Control Area .....	5-32
Enable / Disable Trace Events .....	5-37
Session Overview Area .....	5-39
Daemon Definition Dialog .....	5-41
Import Daemon Definition .....	5-44
General .....	5-46
Target .....	5-46
Trace Events Output .....	5-48
User Trace .....	5-51
Locking Policies .....	5-51
Shared Memory .....	5-53
Timestamp Heartbeat .....	5-53
User Event Buffer .....	5-53
Inheritance .....	5-54
Events .....	5-55
Runtime .....	5-57
Scheduling .....	5-57
CPU Bias .....	5-58
NUMA .....	5-59
Policies .....	5-60
Other .....	5-61
Streaming Options .....	5-61
Kernel Trace Buffer Options .....	5-62



## Using the NightTrace GUI

The NightTrace GUI is invoked using `ntrace` (see “Invoking NightTrace” on page 4-1).

By default, the NightTrace Main Window is presented as shown in the figure below.



**Figure 5-1. NightTrace Main Window**

The NightTrace Main Window consists of the following components:

- NightTrace Main Window Menu Bar (see page 5-3)
- Session Configuration File Name Area (see page 5-31)
- Daemon Control Area (see page 5-32)
- Session Overview Area (see page 5-39)

NightTrace allows users to manage user and kernel NightTrace daemons using *daemon definitions* which are saved as part of the *session* in the session configuration file (see “Session Configuration Files” on page 4-24). These definitions include daemon collection modes and settings, daemon priorities and CPU bindings, and data output formats, as well as the trace event types that are logged by that particular daemon.

Individual daemons within a session may or may not be related to each other in any meaningful way. One might use a session simply to hold several daemon definitions that are commonly used, but not necessarily all at the same time.

Users can manage multiple daemons simultaneously on multiple target systems from a central location and may start, stop, pause, and resume execution of any of the daemons under its management. The user may also view statistics as trace data is being gathered as well as dynamically enable and disable events while a particular daemon is executing.

In addition to sending trace output to a file for later analysis, NightTrace also offers a *streaming* output method. When streaming, trace output is sent directly to the NightTrace display buffer for immediate analysis even while additional trace data is being collected.

## NightTrace Main Window Menu Bar

The NightTrace Main Window menu bar is a part of the NightTrace Main Window (see “Using the NightTrace GUI” on page 5-1).

The NightTrace Main Window menu bar provides access to the following menus:

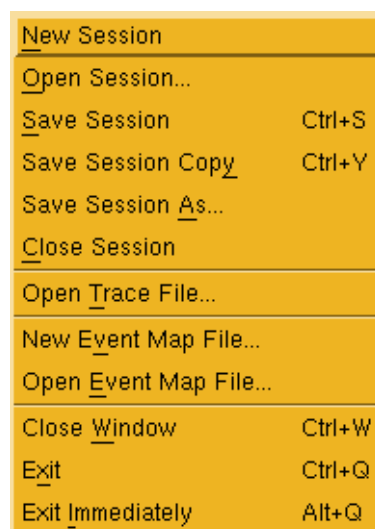
- NightTrace
- Daemons
- Pages
- Options
- Tools
- Help

Each menu is described in the sections that follow.

### NightTrace

The **NightTrace** menu contains session-related items such as initiating a new *session*, saving the current session to a configuration file, and opening a previously-saved configuration file.

The **NightTrace** menu appears on the NightTrace Main menu bar (see “NightTrace Main Window Menu Bar” on page 5-3).



**Figure 5-2. NightTrace menu**

## New Session

Creates a new *session*.

If an existing session is open, it is first closed by this operation.

If changes have been made to the current configuration but have not yet been saved, the **Unsaved Changes** dialog is presented to the user (see “Unsaved Changes” on page 5-7).

## Open Session...

Displays the **Open Session** dialog allowing the user to navigate to the desired directory and select a previously-saved session configuration file to open (see “Session Configuration Files” on page 4-24).

### NOTE

Filename are relative to the *host system* (the system where the NightTrace GUI is running) in the **Open Session** dialog.

If an attempt is made to open a previously-saved session configuration file when changes have been made to the current configuration but have not yet been saved, the **Unsaved Changes** dialog is presented to the user (see “Unsaved Changes” on page 5-7).

## Save Session

**Save Session** saves the current session to a session configuration file (see “Session Configuration Files” on page 4-24 for a complete description of the contents of a session).

**Save Session** allows for quickly saving a session. The user is not prompted for the filenames where the session, trace data, or display pages are to be saved. These are automatically saved in appropriately named files in the current working directory.

If the current session has not been saved to a file in the past, the session is automatically saved to a new session configuration file. The new filename appears in the Session Configuration File Name area above the Daemon Control Area in the NightTrace Main Window (see “Session Configuration File Name Area” on page 5-31).

If the current session was loaded from or previously saved to a session configuration file, the session is saved to that file.

Trace data that has been *touched* is saved by **Save Session**. Touched trace data includes trace data modified by discarding events (see “Discard Events...” on page 9-34). In addition, trace data from a trace data segment file where one or more segments have been saved to another trace data segment file, or closed via the **Close** button in the Session Overview Area (see “Session Overview Area” on page 5-39) is saved.

If the trace data was loaded from a previously saved trace data segment file, the data is saved to that file.

If the trace data has never been saved to a trace data segment file, the data is automatically saved to a newly created trace data segment file

Display pages are saved by **Save Session**. These display pages include those pages created by the **Custom Kernel Page** menu item under the **Pages** menu of the NightTrace Main Window (see “Custom Kernel Page...” on page 5-19) as well as any modified pages.

If the display page was loaded from a previously saved display page file, the page is saved to that file.

If the display page has never been saved to a display page file, the page is automatically saved to a newly created display page file

### **Save Session Copy**

**Save Session Copy** saves the current session to a newly created session configuration file (see “Session Configuration Files” on page 4-24 for a complete description of the contents of a session).

In addition, all trace data is saved to a newly created trace data segment file that becomes part of the newly created session. The trace data itself is now loaded from the newly saved trace data segment file in future invocations of the session. The original trace data sources are left untouched.

The following types of display pages are saved to a newly created display page file that becomes part of the newly created session:

- pages loaded from a file
- pages created by the Custom Kernel Page menu item under the Pages menu of the NightTrace Main Window (see “Custom Kernel Page...” on page 5-19)
- pages modified by the user

**Save Session Copy** allows for quickly saving one or more copies of a session at certain stages. The user is not prompted for the filenames where the session, trace data, or display pages are to be saved. These are saved in appropriately named files in the current working directory.

### **Save Session As...**

Displays the **Save Session** dialog allowing the user to navigate to the desired directory and specify the name of the file to which the session configuration will be saved (see “Session Configuration Files” on page 4-24).

#### **NOTE**

Filenames are relative to the *host system* (the system where the NightTrace GUI is running) in the **Save Session** dialog.

The name of this file will then appear in the Session Configuration File Name Area (see “Session Configuration File Name Area” on page 5-31).

### **Close Session**

Closes the current session but leaves the NightTrace running.

If changes have been made to the current configuration but have not yet been saved, the **Unsaved Changes** dialog is presented to the user (see “Unsaved Changes” on page 5-7).

### **Open Trace File...**

Presents the user with a standard file selection dialog so that they may select a trace event file to load. The event file can be a user trace data file or a kernel trace data file.

### **New Event Map File...**

Presents the user with a standard file selection dialog to select a filename for the new event map file. NightTrace then launches the editor defined by the `EDITOR` environment variable so the user may populate the new event map file with ASCII names for specific trace event values. The file initially contains a template that describes the format of event map file.

See “Event Map Files” on page 4-10 for more information.

### **Open Event Map File...**

Presents the user with a standard file selection dialog to select an event map file to load. An event map file provides ASCII names for specific trace event values.

See “Event Map Files” on page 4-10 for more information.

### **Close Window**

Closes the NightTrace Main Window but leaves any remaining display pages open.

### **Exit**

Closes the session and exits NightTrace completely.

If changes have been made to the current configuration but have not yet been saved, the **Unsaved Changes** dialog is presented to the user (see “Unsaved Changes” on page 5-7).

### **Exit Immediately**

Closes the session and exits NightTrace without prompting to save changes that have been made. Any changes will be lost.



## Unsaved Changes

The **Unsaved Changes** dialog is presented whenever the user attempts to close the session without saving changes that have been made.

### NOTE


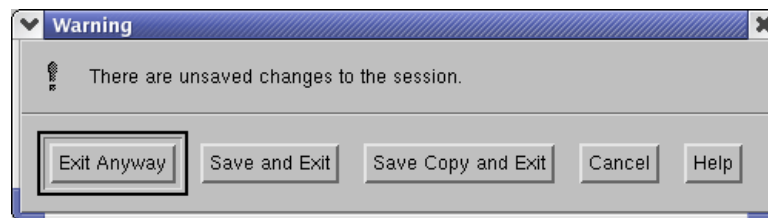
The symbol  appears in the Session Overview area of the NightTrace Main Window as well on display pages. It indicates that there are unsaved changes to either the display page or the trace data segment.

Figure 5-3 shows the dialog that is presented whenever the user attempts to exit NightTrace without saving changes that have been made.



**Figure 5-3. Unsaved Changes / Exit dialog**

### NOTE

To avoid the warning concerning unsaved changes to NightTrace when exiting, the user may choose the **Exit Immediately** menu item (see “Exit Immediately” on page 5-6). However, any unsaved changes will be lost.

### Exit Anyway

Discard the unsaved changes and exit NightTrace.

### Save and Exit

Save the unsaved changes and exit.

If the session had not been saved to a session configuration file (see “Session Configuration Files” on page 4-24) before, the name of the newly created session configuration file is presented in a dialog before exiting.

### Save Copy and Exit

Save the entire session, along with unsaved changes, to a new session configuration file (see “Session Configuration Files” on page 4-24).

Trace data is saved to a newly created trace data file associated with the session (see “Save Session Copy” on page 5-5 for more details).

The name of the newly created session configuration file is presented in a dialog before exiting.

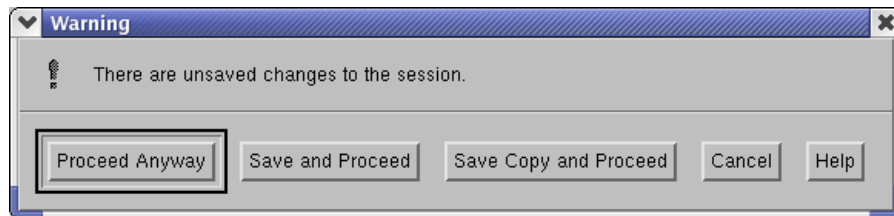
### Cancel

Aborts the exit request.

### Help

Displays the online help for this dialog.

Figure 5-4 shows the dialog that is presented whenever the user attempts to close the current session without exiting (performing a session operation such as **New Session**, **Open Session**, or **Close Session**).



**Figure 5-4. Unsaved Changes / Proceed dialog**

### Proceed Anyway

Discard the unsaved changes and proceed with session operation.

### Save and Proceed

Save the unsaved changes and proceed with the session operation.

If the session had not been saved to a session configuration file (see “Session Configuration Files” on page 4-24) before, the name of the newly created session configuration file is presented in a dialog before proceeding.

**Save Copy and Proceed**

Save the entire session, along with unsaved changes, to a new session configuration file (see “Session Configuration Files” on page 4-24).

Trace data is saved to a newly created trace data file associated with the session (see “Save Session Copy” on page 5-5 for more details).

The name of the newly created session configuration file is presented in a dialog before proceeding.

**Cancel**

Aborts the session operation.

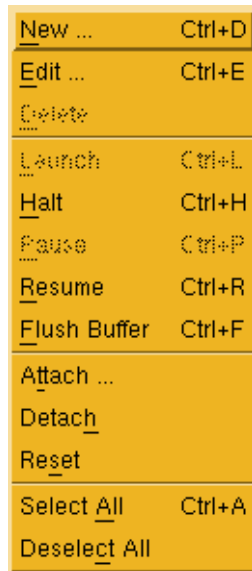
**Help**

Displays the online help for this dialog.

## Daemons

The **Daemons** menu provides functionality for configuring new and existing daemon definitions, as well as attaching to and detaching from running daemons.

The **Daemons** menu appears on the NightTrace Main Window menu bar (see “NightTrace Main Window Menu Bar” on page 5-3).



**Figure 5-5. Daemons menu**

### **New...**

Accelerator: Ctrl+D

Opens the **Daemon Definition** dialog (see “**Daemon Definition Dialog**” on page 5-41) allowing the user to configure a new daemon definition.

### **Edit...**

Accelerator: Ctrl+E

Opens the **Daemon Definition** dialog (see “**Daemon Definition Dialog**” on page 5-41) for the daemon definition currently selected in the **Daemon Control Area** (see “**Daemon Control Area**” on page 5-32) allowing the user to edit that particular definition.

**NOTE**

The daemon definition may not be altered while the daemon is executing.

**Delete**

Deletes the daemon definition(s) currently selected in the Daemon Control Area (see “Daemon Control Area” on page 5-32).

The user is prompted for confirmation before the deletion is performed.

**Launch**

Accelerator: **Ctrl+L**

Starts execution of the daemon(s) currently selected in the Daemon Control Area.

**NOTE**

Starting a daemon does not imply that the daemon begins to collect events.

Launch operations are time consuming and involve possibly connecting to a target system, user authentication, etc. Once the daemon is launched, it is more efficient to utilize the **Pause** and **Resume** operations which require less time and resources.

The same action is performed by pressing the **Launch** button in the Daemon Control Area (see “Launch” on page 5-34).

**Halt**

Accelerator: **Ctrl+H**

Stops execution of the daemon(s) currently selected in the Daemon Control Area.

The connection to the target system is terminated by this operation. Once the daemon is launched, it may be more efficient to utilize the **Pause** and **Resume** operations.

The same action is performed by pressing the **Halt** button in the Daemon Control Area (see “Halt” on page 5-35).

**Pause**

Accelerator: **Ctrl+P**

Pauses the execution of the daemon(s) currently selected in the Daemon Control Area.

## NOTE

When a daemon is paused, incoming trace events are discarded without notice.

The same action is performed by pressing the **Pause** button in the Daemon Control Area (see "Pause" on page 5-35).

### Resume

Accelerator: Ctrl+R

Resumes execution of the daemon(s) currently selected in the Daemon Control Area. Once resumed, incoming events are placed into the daemon buffer for subsequent processing by the daemon.

The same action is performed by pressing the **Resume** button in the Daemon Control Area (see "Resume" on page 5-35).

### Flush Buffer

Accelerator: Ctrl+F

Flushes trace events from the buffers associated with the daemon(s) currently selected in the Daemon Control Area to either the NightTrace display buffer (see "Stream" on page 5-48) or to the output file (see "Output File" on page 5-49).

The same action is performed by pressing the **Flush** button in the Daemon Control Area (see "Flush" on page 5-35).

### Attach...

Allows the user to query any target system for user application trace daemons and displays the results in the **Attach Daemons** dialog (see "Attach Daemons" on page 5-16). The user may then attach to the desired daemon and control it.

### Detach

Relinquishes control of the running daemon(s) currently selected in the Daemon Control Area (see "Daemon Control Area" on page 5-32).

### Reset

Flushes the contents of trace buffers for the running daemon(s) currently selected in the Daemon Control Area (see "Daemon Control Area" on page 5-32). Any events in the buffer at the time of the reset are discarded. Events that have already been written to the output device (file or stream) are unaffected.

Pressing the **Reset** button also places the selected daemons in a **Paused** state (see "State" on page 5-33).

**NOTE**

This option is not supported for kernel trace daemons.

**Select All**

Accelerator: Ctrl+A

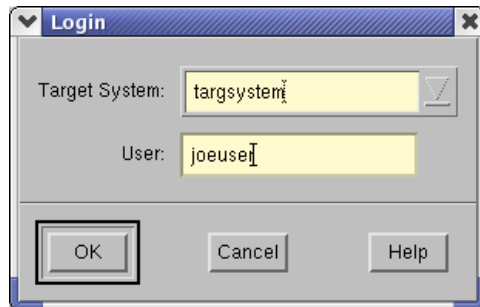
Selects all daemon definitions listed in the Daemon Control Area (see “Daemon Control Area” on page 5-32).

**Deselect All**

Deselects all daemon definitions listed in the Daemon Control Area (see “Daemon Control Area” on page 5-32)

## Login

This dialog is presented when attaching to a daemon on a remote system (see “Attach Daemons” on page 5-16) or when importing daemon attributes based on a user application running on a remote system (see “Import Daemon Definition” on page 5-44).



**Figure 5-6. Login dialog**

After filling in the required fields in the Login dialog, the Enter Password dialog (see “Enter Password” on page 5-14) is displayed, allowing the user to enter the password for the specified User on the specified Target System.

### NOTE

Passwords are not included in the configuration files written by NightTrace. They are retained only during the current invocation of NightTrace.

### Target System

The name of the target system to which the user wishes to connect.

### User

The login name of the user on the specified Target System.

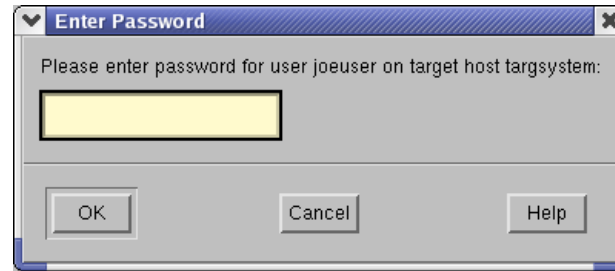
## Enter Password

The Enter Password dialog is displayed during user authentication on a target system.

### NOTE

The Enter Password dialog is not displayed if a valid password has already been entered for the specified user on the specified target system during the current invocation of NightTrace.





**Figure 5-7. Enter Password dialog**

Enter the password for the specified user on the specified target system.

**NOTE**

Passwords are not included in the configuration files written by NightTrace. They are retained only during the current invocation of NightTrace.

## Attach Daemons

The **Attach Daemons** dialog is displayed when the user attempts to attach to a daemon running on a remote target system.

This dialog is presented following user authentication (see “Login” on page 5-14 and “Enter Password” on page 5-14) on that system.

### Figure 5-8. Attach Daemons dialog

#### Program ID

The process ID (PID) of the user trace daemon on the remote system.

#### Creator

The login name of the user who owns the user trace daemon on the remote system.

#### Attach as User

The login name of the user attaching to the user trace daemon. This value defaults to the user specified in the **Login** dialog (see “Login” on page 5-14) presented prior to this dialog.

#### Key File

The filename which is used to calculate the shared memory segment identifier associated with the logging of user trace events. See “Key File” on page 5-48 for more information.

The following buttons appear at the bottom of the **Attach Daemons** dialog and have the specified meaning:

**Attach**

Attaches to the daemon selected in the list and closes the **Attach Daemons** dialog.

**Set Attach as User...**

Brings up a dialog allowing the user to specify the login name used to attach to the selected daemon(s). Since the daemon's shared memory is owned by the creator, the user attaching to the user trace daemon could be relevant in terms of permissions.

**Refresh**

Queries the target system for active trace daemons.

**Cancel**

Closes the **Attach Daemons** dialog without attaching to any of the listed daemons.

**Help**

Provides online help for this dialog.

## Pages

The **Pages** menu allows the user to open preconfigured display pages as well as empty display pages. There is also an option for the user to open up a pre-existing display page.

The **Pages** menu appears on the NightTrace Main Window menu bar (see “NightTrace Main Window Menu Bar” on page 5-3).



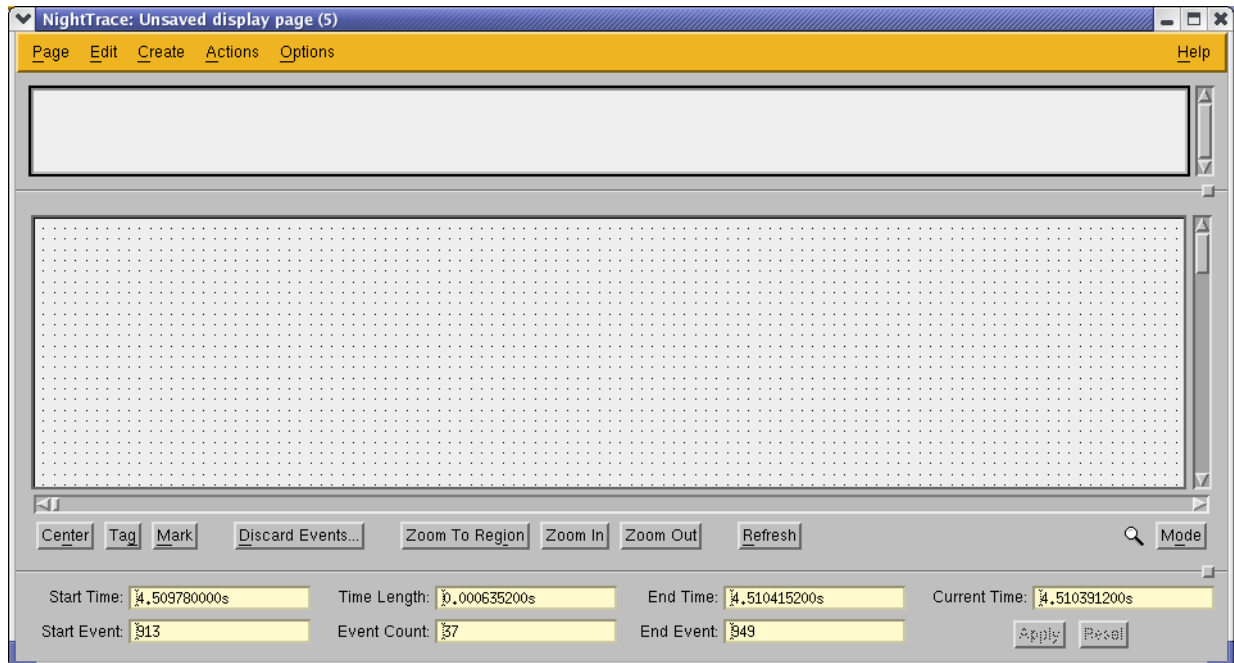
**Figure 5-9. Pages menu**

### New Blank Page

This menu choice opens a new display page (see Chapter 9 “Display Pages”) so that the user may configure it from scratch. The Grid (see must be populated with display objects (see Chapter 10 “Display Objects”) before trace information can be analyzed or graphically examined.

#### NOTE

The new display page comes up in *edit mode* so that display objects may be created and configured (see “Mode Button” on page 9-36 for more information).



**Figure 5-10. New Display Page**

### New User Trace Page

This menu choice opens the default application trace page which is automatically pre-configured to show all user events and specific descriptions of the event ID and the first argument of each event.

See “Default Display Page” on page 9-1 for more information.

### New Ada Trace Page

### Custom Kernel Page...

Presents the **Build Custom Kernel Page** dialog (see “Build Custom Kernel Page” on page 5-21) to quickly build a customized kernel page based on choices of nodes, CPUs, and graphs. When loading kernel trace events in NightTrace, default kernel display pages are displayed for each node where trace data originated. These pages show each CPU for each node, as well as a fixed number of graphs and data boxes per CPU.

However, there may be cases where the default display page for kernel data is not desirable:

- on multi-CPU nodes, the vertical height of the default kernel page may be too large

- when shielding a CPU, or running a process with a CPU bias, it may be desirable to see only data for that CPU
- one or more of the default graphs per CPU may not be of interest

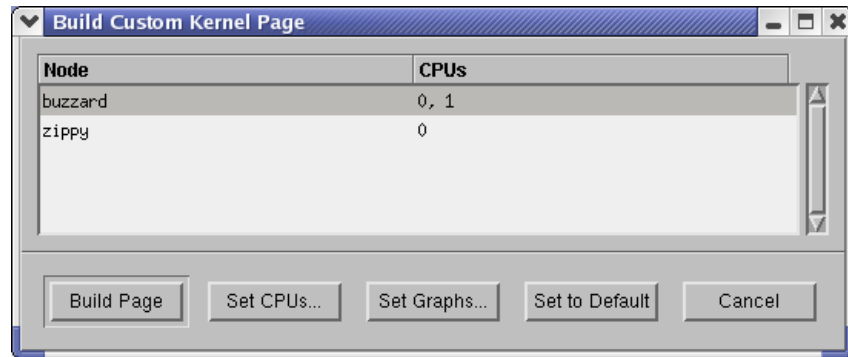
See “Kernel Display Pages” on page 13-6 for more information.

**Open Existing Page...**

This menu choice presents the user with a standard file selection dialog so that they may select a pre-existing configuration page from a previous NightTrace session.

## Build Custom Kernel Page

The Build Custom Kernel Page dialog is opened by selecting Custom Kernel Page... from the Pages menu of the NightTrace Main window (see “Custom Kernel Page...” on page 5-19).



**Figure 5-11. Build Custom Kernel Page dialog**

Select which nodes you would like included on the customized kernel page from the list.

### NOTE

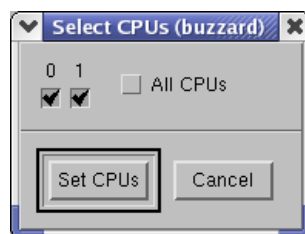
To select multiple items, press the Ctrl key while selecting individual items in the list or hold the Shift key to select a range of items.

### Build Page

Creates the customized kernel page based on choices of nodes, CPUs, and graphs.

### Set CPUs...

Presents the Select CPUs dialog as shown in Figure 5-12 allowing the user to choose which CPUs from the selected node to display in the display page.



**Figure 5-12. Select CPUs dialog**

Select the desired CPUs and press the **Set CPUs** button. **Cancel** dismisses the dialog without making any changes.

Note for each node, the CPUs that would be displayed in the new kernel display page are shown in the **CPUs** column of the **Build Custom Kernel Page** dialog.

**NOTE**

By default, all CPUs per node are displayed by default in the built kernel display page.

**Set Graphs...**

Presents the **Select Graphs** dialog (see “Select Graphs” on page 5-23) allowing the user to choose which graphs to display for each CPU in the display page to be built.

**Set to Default**

Restores the default settings so that all graphs are displayed for all CPUs.

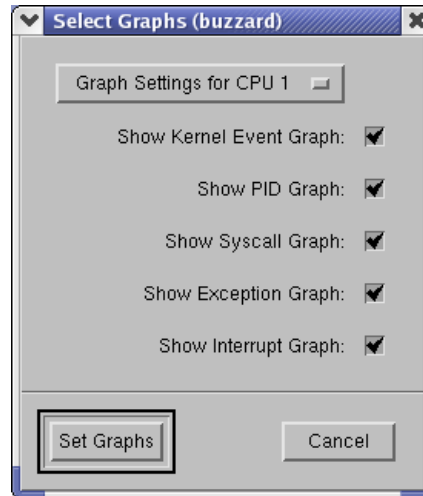
**Cancel**

Aborts the building of the customized kernel display page.



## Select Graphs

The **Select Graphs** dialog allows the user to choose which graphs to display for each CPU in the display page to be built.



**Figure 5-13. Select Graphs dialog**

Select the CPU to which these **Graph Settings** will apply from the drop-down at the top of the dialog and press the **Set Graphs** button after selecting the desired graphs. **Cancel** aborts the selection.

### Show Kernel Event Graph

When this item is checked, the display page will include a kernel event graph for the CPU(s) selected from the **Graph Settings** drop-down.

### Show PID Graph

When this item is checked, the display page will include a PID graph for the CPU(s) selected from the **Graph Settings** drop-down.

See “Running Process Information” on page 13-8 for more information.

### Show Syscall Graph

When this item is checked, the display page will include a syscall graph for the CPU(s) selected from the **Graph Settings** drop-down.

See “Syscall Information” on page 13-12 for more information.

### **Show Exception Graph**

When this item is checked, the display page will include an exception graph for the CPU(s) selected from the **Graph Settings** drop-down.

See “Exception Information” on page 13-10 for more information.

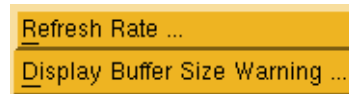
### **Show Interrupt Graph**

When this item is checked, the display page will include an interrupt graph for the CPU(s) selected from the **Graph Settings** drop-down.

See “Interrupt Information” on page 13-9 for more information.

## Options

The Options menu appears on the NightTrace Main Window menu bar (see “NightTrace Main Window Menu Bar” on page 5-3).



**Figure 5-14. Options menu**

### **Refresh Rate...**

Displays the Refresh Interval dialog (see “Refresh Interval” on page 5-26) allowing the user to specify how often the statistics (displayed in the Daemon Control Area) are requested and updated for running daemons. This dialog sets the display rate for those daemons currently selected in the Daemon Control Area (see “Daemon Control Area” on page 5-32).

### **Display Buffer Size Warning...**

Presents the Display Buffer Size Warning dialog (see “Display Buffer Size Warning” on page 5-27) allowing the user to specify a limit for the amount of memory used to hold trace data before a warning is issued. This dialog also allows the user to instruct NightTrace to halt any active daemons when this limit is reached.

## Refresh Interval

This dialog allows the user to specify how often the statistics (displayed in the Daemon Control Area) are requested and updated for running daemons. This dialog sets the display rate for those daemons currently selected in the Daemon Control Area (see “Daemon Control Area” on page 5-32).

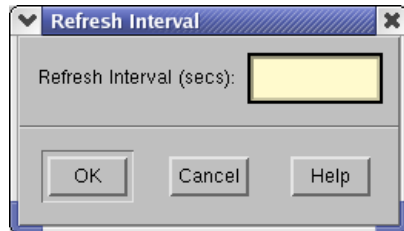


Figure 5-15. Refresh Interval dialog

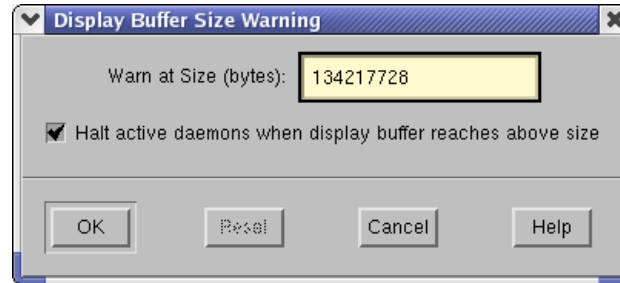
### Refresh Interval

The number of seconds between queries.

## Display Buffer Size Warning

When *streaming* trace data directly from the trace daemons to NightTrace (see “Stream” on page 5-48), significant amounts of memory can be used quickly. To alleviate this problem, a *display buffer* is used to hold the available trace data.

This dialog allows the user to specify a limit on the amount of memory used by the display buffer before a warning is issued. In addition, the user can select to halt active daemons when the display buffer reaches the specified limit.



**Figure 5-16. Display Buffer Size Warning dialog**

### Warn at Size

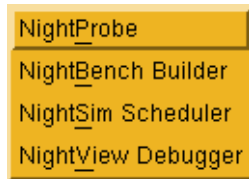
The number of bytes used by the display buffer before a warning is issued.

### Halt active daemons when display buffer reaches above size

If checked, all active daemons in the current session will be halted when the size of the display buffer reaches the limit specified in this dialog (see Warn at Size).

## Tools

The **Tools** menu appears on the NightTrace Main Window menu bar (see “NightTrace Main Window Menu Bar” on page 5-3).



**Figure 5-17. Tools menu**

### NightProbe

Opens the NightProbe Data Monitoring application. NightProbe is a real-time graphical tool for monitoring, recording, and altering program data within one or more executing programs without significant intrusion. NightProbe can be used in a development environment as a tool for debugging, or in a production environment to create a “control panel” for program input and output.

See also:

- *NightProbe User's Guide* (0890480)

### NightBench Builder

Opens the NightBench Program Development Environment. NightBench is a set of graphical user interface (GUI) tools for developing software with the Concurrent C/C++ and MAXAda™ compiler toolsets.

#### NOTE

NightBench is currently not available on RedHawk systems.

See also:

- *NightBench User's Guide* (0890480)

### NightSim Scheduler

Opens the NightSim Application Scheduler. NightSim is a tool for scheduling and monitoring real-time applications which require predictable, repetitive process execution. With NightSim, application builders can control and dynamically adjust the

periodic execution of multiple coordinated processes, their priorities, and their CPU assignments.

See also:

- *NightSim User's Guide* (0890480)

### **NightView Debugger**

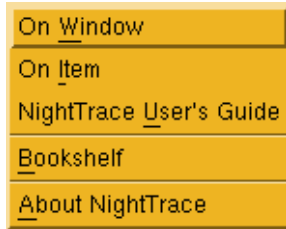
Opens the NightView Source-Level Debugger. NightView is a graphical source-level debugging and monitoring tool specifically designed for real-time applications. NightView can monitor, debug, and patch multiple real-time processes running on multiple processors with minimal intrusion.

See also:

- *NightView User's Guide* (0890395)

## Help

The **Help** menu appears on the NightTrace Main Window menu bar (see “NightTrace Main Window Menu Bar” on page 5-3).



**Figure 5-18. Help menu**

### **On Window**

Displays the help topic for the current window.

### **On Item**

Gives context-sensitive help on the various menu options, dialogs, or other parts of the user interface.

Help for a particular item is obtained by first choosing the **On Item** menu option, then clicking the mouse pointer on the object for which help is desired (the mouse pointer will become a floating question mark when the **On Item** menu item is selected).

In addition, context-sensitive help may be obtained for the currently highlighted option by pressing the F1 key. The HyperHelp viewer will display the appropriate topic.

### **NightTrace User's Guide**

Opens the online *NightTrace User's Guide*.

### **Bookshelf**

Opens a HyperHelp window that lists all of the Concurrent online publications currently available on the local system.

### **About NightTrace**

Displays version and copyright information for the NightTrace product.



## Session Configuration File Name Area

The area located directly beneath the NightTrace Main Window Menu Bar displays the name of the current session configuration file (see “Session Configuration Files” on page 4-24).



**Figure 5-19. Session Configuration File Name Area**

### Session configuration file

The name of the current session configuration file. If the current session configuration has not yet been saved to a file, **New** will be displayed in this area.

To save the current session configuration to a file, select one of the following menu items:

- **Save Session** (see “Save Session” on page 5-4)
- **Save Session Copy** (see “Save Session Copy” on page 5-5)
- **Save Session As...** (see “Save Session As...” on page 5-5)

from the NightTrace menu (see “NightTrace” on page 5-3).

## Daemon Control Area

The area located directly beneath the Session Configuration File Name Area displays information about the daemons defined in the current session.

Double-clicking on an entry in the Daemon Control Area brings up the Daemon Definition Dialog for the daemon associated with that entry (see “Daemon Definition Dialog” on page 5-41).

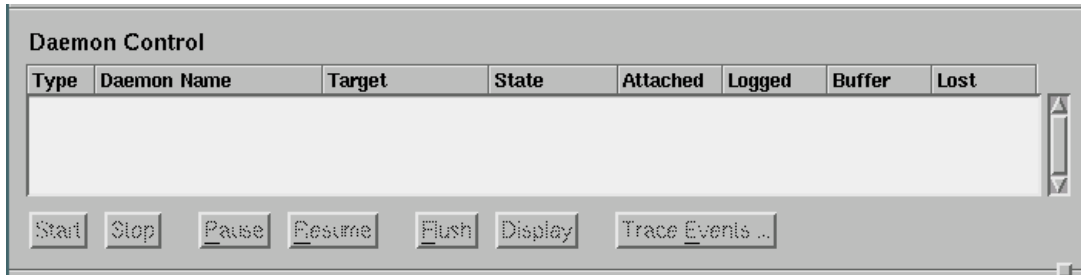


Figure 5-20. Daemon Control Area

### Type

Indicates what type of trace events the daemon is logging.

- U indicates that the associated daemon is logging user trace events
- K indicates that the associated daemon is logging kernel trace events

The type of trace event that the daemon is logging is configured by selecting either the Kernel or the User Application radiobutton in the Trace section on the General page of the Daemon Definition dialog (see “General” on page 5-46).

### Daemon Name

The name of the daemon as configured in the Name field on the General page of the Daemon Definition dialog (see “Name” on page 5-46).

### NOTE

The Daemon Name is merely a label to aid the user in identifying specific daemons with a session. It has no external meaning and is unrelated to the NightTrace API.

**Target**

The name of the system on which the associated daemon is running.

The target system is specified in the **Target System** field on the **General** page of the **Daemon Definition** dialog (see “Target System” on page 5-47).

**State**

The state of the daemon.

<b>Logging</b>	indicates the daemon is currently capturing events
<b>Halted</b>	indicates the daemon is not executing
<b>Paused</b>	indicates the daemon is started but is not capturing events  While paused, attempts to log events from user applications or via the operating system kernel are discarded. Note that these are not considered lost events (see “Lost” on page 5-34).
<b>Pausing</b>	indicates the daemon is going from a <b>Logging</b> state to a <b>Paused</b> state
<b>Resuming</b>	indicates the daemon is going from a <b>Paused</b> state to a <b>Logging</b> state
<b>Lauching</b>	indicates the daemon is going from a <b>Halted</b> state to a <b>Logging</b> state
<b>Halting</b>	indicates the daemon is going from a <b>Paused</b> or <b>Logging</b> state to a <b>Halted</b> state

**Attached**

The number of user application threads or processes that are associated with the daemon.

**Streaming**

Indicates whether or not data from this daemon is being streamed to the NightTrace display buffer. This is specified by the setting of the **Stream** checkbox on the **General** page of the **Daemon Definition** dialog (see “Stream” on page 5-48).

Advanced settings with respect to *streaming* can be found on the **Other** page of the **Daemon Definition** dialog (see “Other” on page 5-61).

If streaming is not in effect, data will be written to the output file (see “Output File” on page 5-49) as specified on the **General** page of the **Daemon Definition** dialog.

### **Logged**

The number of trace events that have been written to the stream or written to the file by the associated daemon. See **Streaming** above.

### **Buffer**

The number of trace events currently held in the buffer.

These events will be flushed from the buffer either when the **Flush Threshold** (see “Flush Threshold” on page 5-54) is reached or when the user flushes them manually (see “Flush” on page 5-35).

### **Lost**

Lost events occur when the daemon cannot keep up with the rate at which events are being added to the buffer.

To combat this, adjust the **Runtime** attributes of the daemon by raising its **Priority** and/or by changing its **CPU Bias** to bind it to a specific CPU. (See “Runtime” on page 5-57 for a description of these settings.)

### **NOTE**

Events that are discarded when a daemon is **Paused** (see “State” on page 5-33) are not included in the **Lost** count.

Also, events that are discarded when the daemon is in **Buffer Wrap** mode (see “Buffer Wrap” on page 5-50) (i.e. older events being discarded in favor of new ones) are not included in the **Lost** count.

The area located at the bottom of the Daemon Control Area contains a number of buttons which control the daemons currently selected in the Daemon Control Area.

### **Launch**

Accelerator: **Ctrl+L**

Starts execution of the daemon(s) currently selected in the Daemon Control Area.

### **NOTE**

Starting a daemon does not imply that the daemon begins to collect events.

Launch operations are time consuming and involve possibly connecting to a target system, user authentication, etc. Once the daemon is launched, it is more efficient to utilize the **Pause** and **Resume** operations which require less time and resources.

### **Halt**

Accelerator: Ctrl+H

Stops execution of the daemon(s) currently selected in the Daemon Control Area.

The connection to the target system is terminated by this operation. Once the daemon is launched, it may be more efficient to utilize the **Pause** and **Resume** operations.

### **Pause**

Accelerator: Ctrl+P

Pauses the execution of the daemon(s) currently selected in the Daemon Control Area.

### **NOTE**

When a daemon is paused, incoming trace events are discarded without notice.

### **Resume**

Accelerator: Ctrl+R

Resumes execution of the daemon(s) currently selected in the Daemon Control Area. Once resumed, incoming events are placed into the daemon buffer for subsequent processing by the daemon.

### **Flush**

Accelerator: Ctrl+F

Flushes trace events from the buffers associated with the daemon(s) currently selected in the Daemon Control Area to either the NightTrace display buffer (see “Stream” on page 5-48) or to the output file (see “Output File” on page 5-49).

### **Display**

When data from the selected daemon(s) is being streamed to the NightTrace display buffer (as specified by the setting of the **Stream** checkbox on the **General** page of the **Daemon Definition** dialog (see “General” on page 5-46)), pressing this button causes a flush of the data currently in the trace buffer to the NightTrace display buffer. If no display pages currently exist, a default display page will be created when this button is pressed.

**NOTE**

The user must scroll the NightTrace display in order to see the most up-to-date data.

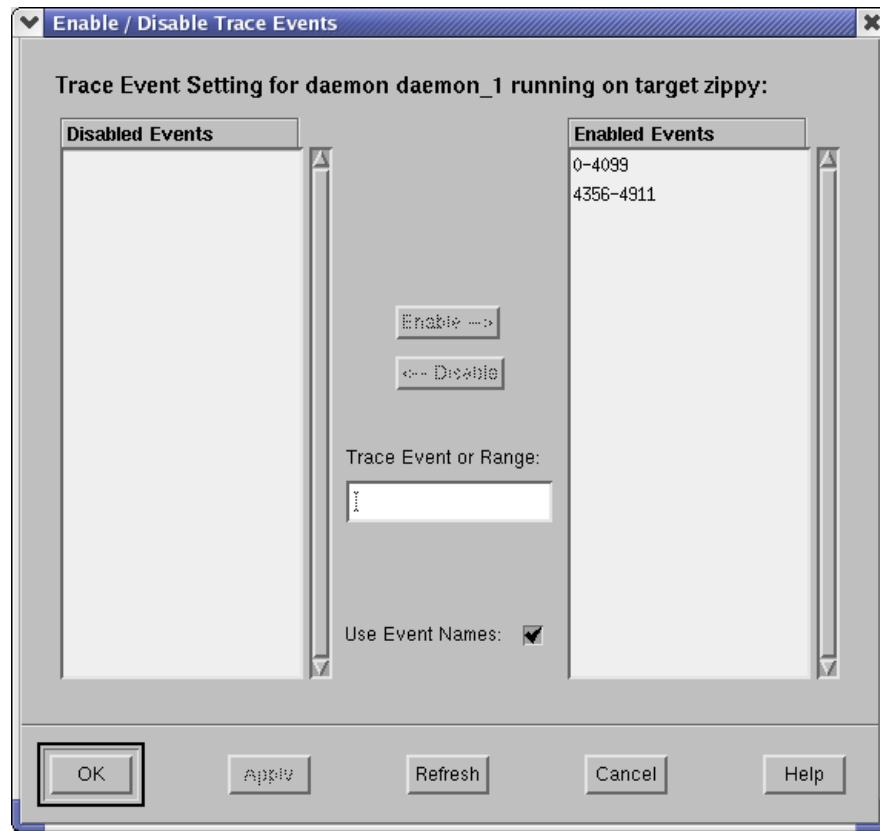
When data from the selected daemon(s) is written to output files, pressing this button causes the data in the output file to be displayed in the NightTrace display.

**Trace Events...**

Presents the **Enable/Disable Trace Events** dialog (see “Enable / Disable Trace Events” on page 5-37) allowing the user to dynamically enable or disable selected trace event types while a particular daemon is running. A currently executing daemon must be selected from the Daemon Control Area.

## Enable / Disable Trace Events

The Enable/Disable Trace Events dialog allows the user to dynamically enable or disable selected trace event types while a particular daemon is running. This dialog is opened by selecting a currently executing daemon from the Daemon Control Area and pressing the Trace Events... button in the Daemon Control Area of the NightTrace Main Window (see “Daemon Control Area” on page 5-32).



**Figure 5-21. Enable / Disable Trace Events dialog**

### Disabled Events

This is a list of user trace or kernel trace event types that are disabled.

Disabled events are not logged to daemon buffers and therefore are not included in event trace outputs.

### Enabled Events

This is a list of user trace or kernel trace event types that are enabled.

Enabled events are allowed to be placed into daemon buffers and are subsequently transferred to the output device (see "Trace Events Output" on page 5-48).

**Enable -->**

Moves the selected items from the Disabled Events list or the Trace Event or Range field to the Enabled Events list.

**<-- Disable**

Moves the selected items from the Enabled Events list or the Trace Event or Range field to the Disabled Events list.

**Trace Event or Range**

Allows the user to enter a particular trace event type (or range of trace event types) and subsequently Enable --> or Disable --> it.

The user may use the event name associated with the event type (e.g. TR\_SYSCALL\_RESUME) or the numerical value of the trace event type (e.g. 4131).

The user may also enter a range of values either using the event names or their numerical values (e.g. TR\_INTERRUPT\_ENTRY-TR\_EXCEPTION\_EXIT or 4112-4117).

**Use Event Names**

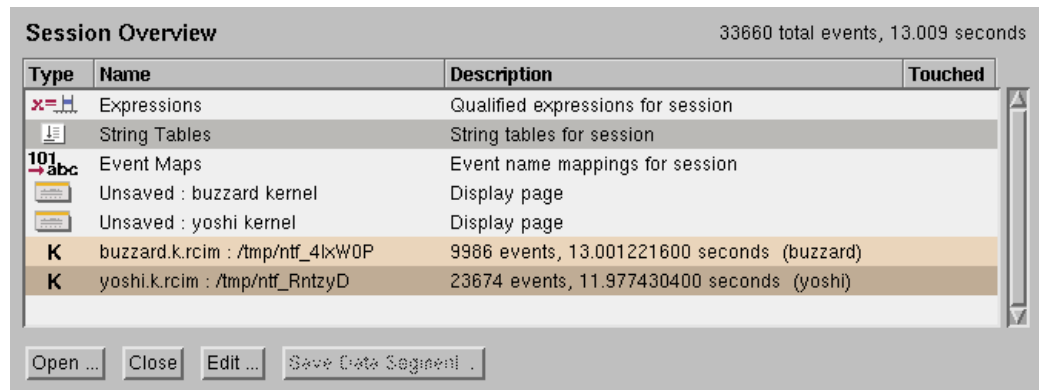
Allows the user to view the event names of the trace event types in the Disabled Events and Enabled Events lists instead of their numerical values.

For user trace events, the user may load user-defined event map files which associate meaningful names with the user trace event ID numbers (see "Event Map Files" on page 4-10).



## Session Overview Area

The Session Overview Area provides an interface where commonly used components of a *session* may be managed.



**Figure 5-22. Session Overview Area**

Each component is described briefly in the Session Overview list:

### Type

This column displays an icon representing the type of item in the list.

<b>K</b>	kernel trace data
<b>U</b>	user trace data
	display pages
	string tables
	event map entries
	qualified expressions


### Name

Name of session component.

### Description

This column describes the item and may include statistics for trace data sets.

### **Touched**

A  in this column indicates if there have been changes to the session component that are not saved.

The area located at the bottom of the Session Overview Area contains a number of buttons which apply to the files currently selected in the Session Overview Area.

### **Open...**

Presents the user with a standard file selection dialog which allows them to open trace data files, trace configuration files, or trace event map files.

### **Close**

Closes the selected files from the Session Overview list and removes them from the session. When closing selected files that are **Temporary**, the files are deleted.

### **Edit...**

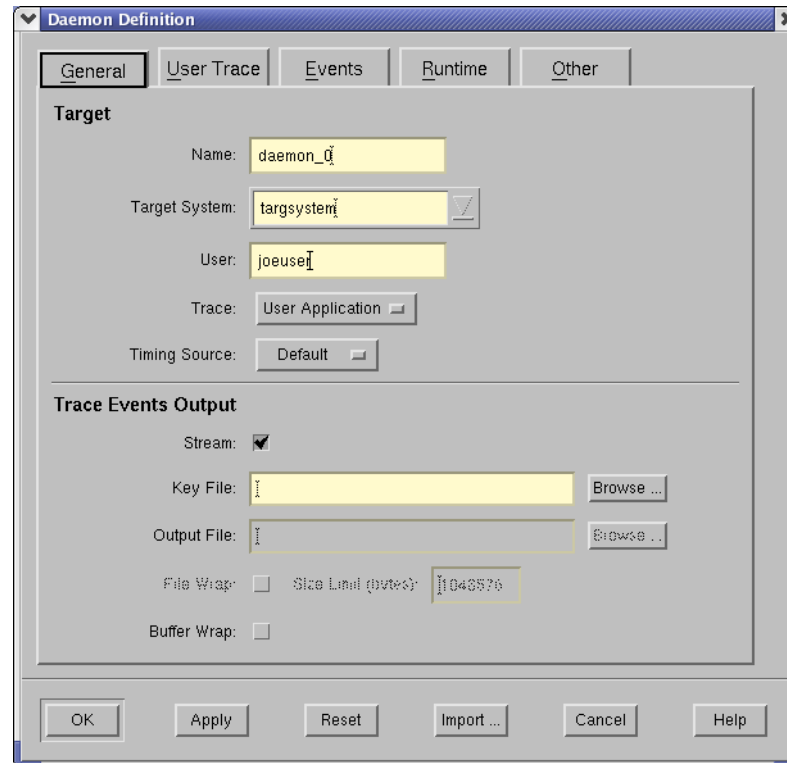
Launches the selected file into an editor allowing the user to search and even modify the selected file. This is most useful for event map files, configuration files containing tables, and kernel vector files which contain system call, interrupt, and exception names.

### **Save Data Segment...**

Presents the user with a standard file selection dialog allowing the user to save trace data in a much more efficient format than raw trace event files providing for faster initialization at startup.

## Daemon Definition Dialog

The Daemon Definition dialog allows the user to create and modify the various aspects of a daemon configuration.



**Figure 5-23. Daemon Definition dialog**

The Daemon Definition dialog is divided into a number of pages that contain specific information about the current configuration. These pages are:

- General

This page contains information such as the name of the daemon configuration, the target system on which the daemon will run, the user's login on that system, and settings specifying whether kernel or user application tracing will be performed. Items related to trace events output such as the names of output and key files and settings such as whether or not *streaming* will be performed by this daemon are found on this page as well.

See "General" on page 5-46 for more detailed information.

- User Trace

This page contains settings for user trace daemons such as locking policies associated with the daemon, shared memory permissions, and the duration of the times-

tamp heartbeat, as well as specifications for the size and flush threshold of the user event buffer.

See “User Trace” on page 5-51 for more detailed information.

- **Events**

This page allows the user to specify which events may be logged while tracing.

See “Events” on page 5-55 for more detailed information.

- **Runtime**

This page allows the user to specify the scheduling policy, CPU bias, and memory binding policies for the daemon.

See “Runtime” on page 5-57 for more detailed information.

- **Other**

This page allows the user to specify advanced settings with respect to the transfer of trace data from the daemon to the NightTrace display buffer.

See “Other” on page 5-61 for more detailed information.

The following buttons appear at the bottom of the **Daemon Definition** dialog and have the specified meaning:

**OK**

This button applies changes made and closes the **Daemon Definition** dialog.

**Apply**

This button applies changes made but leaves the **Daemon Definition** dialog open.

**Reset**

This button restores the values of all items to the previously-applied values and leaves the **Daemon Definition** dialog open.

**Import...**

Presents the **Import Daemon Definition** dialog (see “Import Daemon Definition” on page 5-44) allowing the user to define daemon attributes based on a user application running on a remote system. The **Import Daemon Definition** dialog is presented following user authentication (see “Login” on page 5-14 and “Enter Password” on page 5-14).

**Cancel**

This button restores the values of all items to the previously-applied values and closes the **Daemon Definition** dialog.

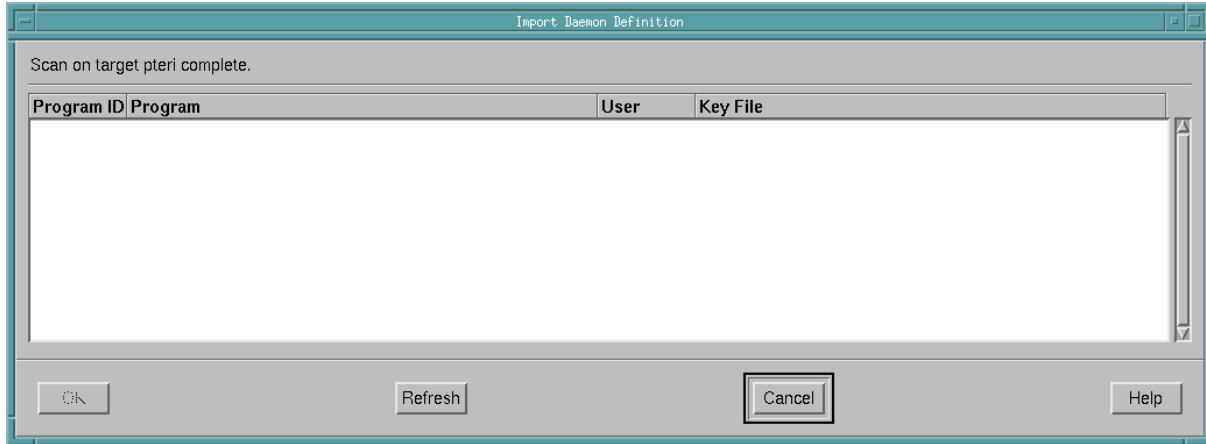
**Help**

This button brings up the help topic for this page.

## Import Daemon Definition

This dialog allows the user to define daemon attributes based on a running user application containing NightTrace API calls. The **Import Daemon Definition** dialog is presented following user authentication (see “Login” on page 5-14 and “Enter Password” on page 5-14).

The user may select an application, running on the specified target system, from which they wish to import trace-related attributes.



**Figure 5-24. Import Daemon Definition dialog**

### Program ID

The process ID (PID) of the Program on the remote system.

### Program

The name of the user application containing `trace_` calls on the remote system.

### User

The user who invoked the Program on the remote system.

### Key File

The filename which is used to calculate the shared memory segment identifier associated with the logging of user trace events. See “Key File” on page 5-48 for more information.

The following buttons appear at the bottom of the **Import Daemon Definition** dialog and have the specified meaning:

**OK**

Imports daemon attributes into the current daemon definition from the user application selected in the list.

**Refresh**

Queries the specified target system for user applications making trace-related calls.

**Cancel**

This button closes the **Import Daemon Definition** dialog without importing any daemon attributes from any of the listed applications.

**Help**

Brings up online help for this dialog.

## General

The **General** page of the **Daemon Definition** dialog (see “Daemon Definition Dialog” on page 5-41) contains information such as the name of the daemon configuration, the target system on which the daemon will run, the user's login on that system, and settings specifying whether kernel or user application tracing will be performed. Items related to trace events output such as the names of output and key files and settings such as whether or not *streaming* will be performed by this daemon are found on this page as well.

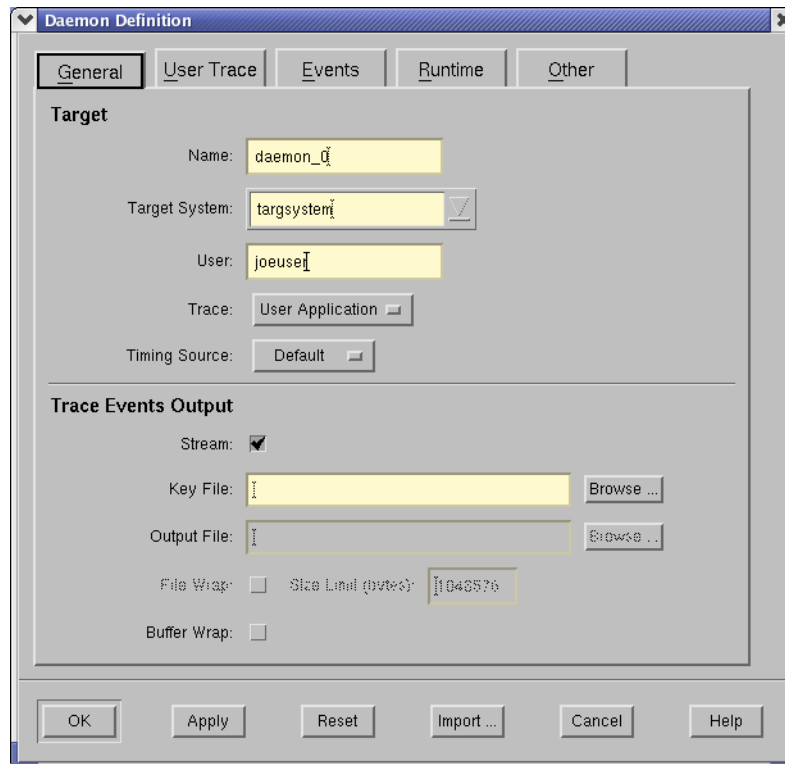


Figure 5-25. Daemon Definition dialog - General

## Target

### Name

The name for this daemon definition.

This field is automatically populated with the name `daemon_x` where `x` is a number, starting at 0, which increments with each new daemon definition.

The **Name** is merely a label to aid the user in identifying specific daemons with a session. It has no external meaning and is unrelated to the NightTrace API. The user may change this to a name of their choosing.



## Target System

The system on which this trace daemon will run.

## User

The name of the user on the specified **Target System** responsible for running this daemon.

## Trace

Indicates what type of trace events this daemon will be logging.

## Kernel

Indicates that the daemon is logging kernel trace events.

Kernel events are automatically generated by the operating system kernel when a kernel daemon is initiated if the operating system kernel was built with tracing support.

See the *PowerMAX OS Real-Time Guide* (0890466) for information on configuring the kernel for kernel tracing on a PowerMAX OS system.

For systems running RedHawk Linux, see the *Concurrent Real-Time Linux - RT User Guide* (0898004) for more detailed information.

## User Application

Indicates that the daemon is logging user trace events.

User trace events are generated by:

- user applications that use the NightTrace API
- the NightProbe tool (see the description of the **To NightTrace** menu item in the chapter titled “Using the Data Recording Window” in the *NightProbe User’s Guide* (0890480)).

## Timing Source

By default, an architecture-specific clock is used to timestamp trace events. On NightHawk 6000 Series machines, the interval timer is used; on Power Hawk and PowerStack systems, it is the PowerPC Time Base Register; on iHawk systems, the Intel Time Stamp Counter is used.

NightTrace can also specify the Real-Time Clock and Interrupt Module (RCIM) as a timestamp source (see “Timestamp Source Selection” on page 1-2 for more information). This is most useful when concurrent traces running on multiple systems are desired. Using the RCIM as a timing device allows NightTrace to present the user with a synchronized view of concurrent activities on those systems.

### Default

Specifies that the architecture-specific clock will be used to timestamp trace events. On NightHawk 6000 Series machines, the interval timer is used; on Power Hawk and PowerStack systems, it is the PowerPC Time Base Register; on iHawk systems, the Intel Time Stamp Counter is used.

### RCIM Tick

Specifies that the Real-Time Clock and Interrupt Module (RCIM) tick clock will be used to timestamp trace events.

### NOTE

Use of this option requires that an RCIM board is installed and configured on the target system.

## Trace Events Output

### Stream

When checked, this specifies that *streaming* is in effect so that the output trace events will go directly to the NightTrace display buffer. Otherwise, the output will be written to the Output File (see below).

### Key File

Specifies a filename which is used to calculate the shared memory segment identifier associated with the logging of user trace events. The daemon and the NightTrace API use the `ftok(2)` service to map the specified filename to a shared memory identifier as used by `shmat(2)`.

### NOTE

When the output method is NOT *streaming* (see **Stream** above), the **Key File** defines the name of the **Output File** where trace events are written (see "Output File" on page 5-49).

The **Key File** is relative to the target system. It does not necessarily need to be accessible from the *host system* (the system where the NightTrace GUI is running); however, that can be convenient for subsequent analysis via NightTrace.

Furthermore, the **Key File** does not have to pre-exist. If a user application has not already created it via a NightTrace API call, the daemon will create the file if it does not exist.

### **Browse...**

Brings up a standard file selection dialog so that the user may navigate to the desired location of the **Key File**.

In order to browse, the **Target System** (see “Target System” on page 5-47) must be operational. The file selection dialog invoked by that button shows files relative to the **Target System**.

### **Output File**

The name of the file to which trace events are written.

The **Output File** is relative to the target system. It does not necessarily need to be accessible from the *host system* (the system where the NightTrace GUI is running); however, that can be convenient for subsequent analysis via NightTrace.

### **NOTE**

When the output method is NOT *streaming* (see **Stream** above), the **Key File** (see “Key File” on page 5-48) defines the name of the **Output File**.

### **Browse...**

Brings up a standard file selection dialog so that the user may navigate to the desired location of the **Output File**.

In order to browse, the **Target System** (see “Target System” on page 5-47) must be operational. The file selection dialog invoked by that button shows files relative to the **Target System**.

### **File Wrap**

When checked, allows the user to specify the **Maximum File Size** for the **Key File/Output File**.

### **Maximum File Size**

The maximum number of bytes for the **Key File/Output File**.

When the **Maximum File Size** is reached, subsequent events will overwrite the oldest events. NightTrace automatically detects this and presents events in chronological order, from oldest to newest. Events that are discarded due to **File Wrap** are NOT considered “lost events” (see “Lost” on page 5-34) in statistics provided by the NightTrace.

## NOTE

For a daemon capturing kernel trace events, the file wrap sizes that the user specifies are rounded up to a multiple of kernel buffer sizes. (On PowerMAX OS systems, a kernel trace buffer has a fixed size of 4096\*12 bytes; on RedHawk systems, a kernel trace buffer is 500000 bytes.)

### Buffer Wrap

When this is checked, the daemon will overwrite the least recently recorded events in the trace buffer when it reaches its maximum size.

For user trace events, the size of the buffer is specified in the **Buffer Size** field on the **User Trace** page of the **Daemon Definition** dialog (see "User Trace" on page 5-51).

For kernel trace events, the size of the buffer is defined by the operating system.

On a PowerMAX OS system, a kernel trace buffer has a fixed size of 4096\*12 bytes which holds 4095 kernel events. The total number of trace buffers for kernel events is specified by the kernel tunable `TR_BUFFER_COUNT`, the default value of which is 5.)

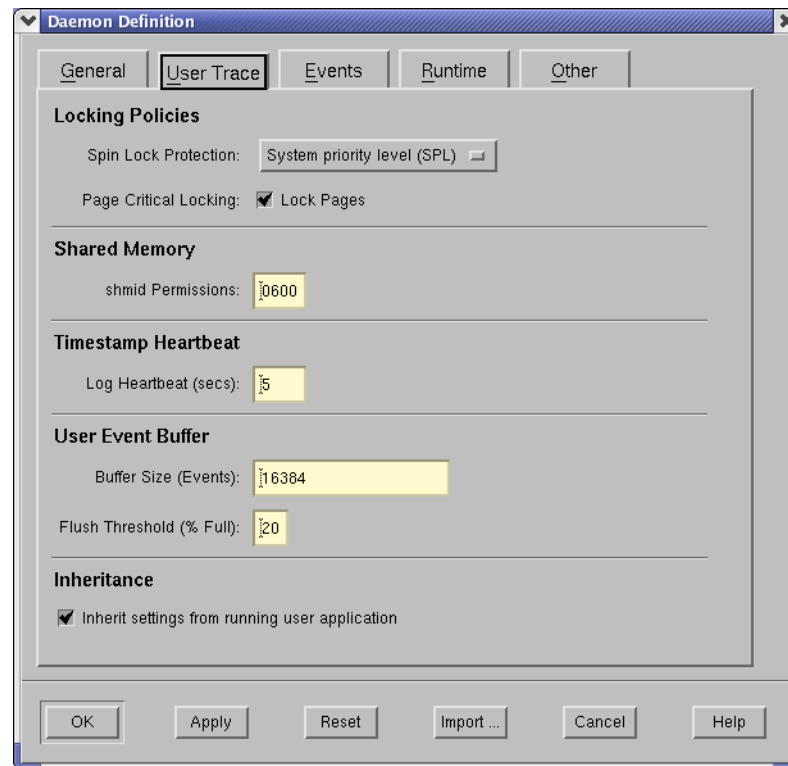
## NOTE

The value of `TR_BUFFER_COUNT` may be changed on a PowerMAX OS system via the `config` command. However, if this tunable is changed, the kernel must be rebuilt and the system restarted for the change to take effect.

On RedHawk systems, there are two kernel trace buffers, each of which is 500000 bytes in size.

## User Trace

The **User Trace** page of the **Daemon Definition** dialog (see “**Daemon Definition Dialog**” on page 5-41) contains settings for locking policies associated with the daemon and the corresponding user applications using the NightTrace API, shared memory permissions, and the duration of the timestamp heartbeat, as well as specifications of the size and flush threshold of the user event buffer.



**Figure 5-26. Daemon Definition dialog - User Trace**

## Locking Policies

### Spin Lock Protection

The NightTrace API and associated daemons use high-performance, low-intrusion spin locks to protect critical sections involved in logging and consuming trace events.

These spin locks require preemption control so that processes on the same CPU don't preempt a daemon or user process in the middle of a critical section and then spin forever waiting for it to be unlocked.

The spin locks are held for extremely short periods of time.

Failure to properly select a protection level may result in a process spinning forever on a CPU in the event of unfortunate preemption.

In more severe cases, the system may hang; this is only a problem if a user-level interrupt preempts another user process or daemon at an unfortunate time and attempts to log trace events to the same trace daemon session.

Data corruption of the trace data will not occur in any case.

### **System priority level (SPL)**

This is the safest form of preemption control as it prevents even machine interrupts from preempting the locking process. This is required when a user application will be logging events in a user-level interrupt handler (i.e. at system interrupt level).

This is the default locking protection mechanism for PowerMAX OS systems.

### **NOTE**

This mechanism is not available on RedHawk systems. If selected for RedHawk systems, it is silently translated to the **Rescheduling Variables** protection as described below.

### **Rescheduling Variables**

This level of protection is sufficient for user applications that log trace events as long as no user-level interrupt handlers will be logging trace events.

### **None**

Selecting no protection opens up the real possibility that the user process or the daemon could preempt each other if they are allowed to operate on the same CPU.

However, the **Runtime** subpage allows the user to define the CPU binding (**CPU Bias**) and priority (**Priority**) at which the daemon operates. (See "Runtime" on page 5-57 for a description of these settings.) Thus, if the user also takes similar care to schedule their user applications then selecting **None** is sufficient.

### **Page Critical Locking**

Page locking is required to prevent preemption while holding a spin lock. Without this choice, it is possible that a page fault could occur while a spin lock is held, allowing for a user application or daemon to spin forever.

### **Lock Pages**

When this option is selected, the daemon and user applications associated with this daemon lock down the required pages and unlock them when the NightTrace API is terminated.

## **Shared Memory**

The daemon and the user applications communicate with each other through shared memory. The shared memory segment identifier is calculated from the **Key File** setting (see “Key File” on page 5-48).

The shared memory segment is automatically destroyed after the last user application and/or the daemon exits normally (if the daemon or user applications are aborted, the shared memory segment will remain; it will be reinitialized on subsequent use).

### **shmid Permissions**

If the daemon is initiated before any user applications, then the shared memory segment will be created with the specified permissions.

## **Timestamp Heartbeat**

For performance reasons, NightTrace events normally include only the low 32 bits of a full 64-bit timestamp. The heartbeat ensures that the daemon logs a full 64-bit timestamp before the interval of time represented by 32-bits expires. The daemon automatically calculates the heartbeat time when it determines how fast the timing source clock ticks.

### **Log Heartbeat**

The frequency at which a full 64-bit timestamp will be generated.

### **NOTE**

There would be no particular benefit by setting the heartbeat to a value shorter than the automatically calculated time unless trace time anomalies are seen because the daemon is preempted by higher priority processing and cannot otherwise log the heartbeat in time.

## **User Event Buffer**

### **Buffer Size**

The number of events that can be held in memory before being written to the output device.

## Flush Threshold

The **Flush Threshold**, expressed as a percentage of the **Buffer Size**, is the point at which the daemon begins to transfer events from the user event buffer to the output device (see “Trace Events Output” on page 5-48). The threshold is important so as not to lose events if the daemon cannot respond quickly enough.

### NOTE

If events are being lost, a combination of changing the **Buffer Size**, the **Flush Threshold**, and the runtime **Priority** (see “Priority” on page 5-58) of the daemon is normally sufficient to prevent event loss.

## Inheritance

When the daemon starts up, certain settings can be inherited from a running user application that has set up a trace definition.

The NightTrace API `trace_begin()` call (an enhanced replacement for `trace_start()`) allows the user to define the following settings in a user application:

- those values listed under the **Spin Lock Protection** and **Page Critical Locking** categories on this page
- the **Buffer Size** also found on this page
- the setting for the **Timing Source** which appears on the **General** page of the **Daemon Definition** dialog (see “General” on page 5-46)

See “`trace_begin()`” on page 2-5 for more information on this API.

### Inherit settings from running user application

When this is checked, trace settings defined by a running user application are silently preferred if those definitions differ from those made in NightTrace.

If not checked, trace settings defined by user applications must match those in the current daemon definition.

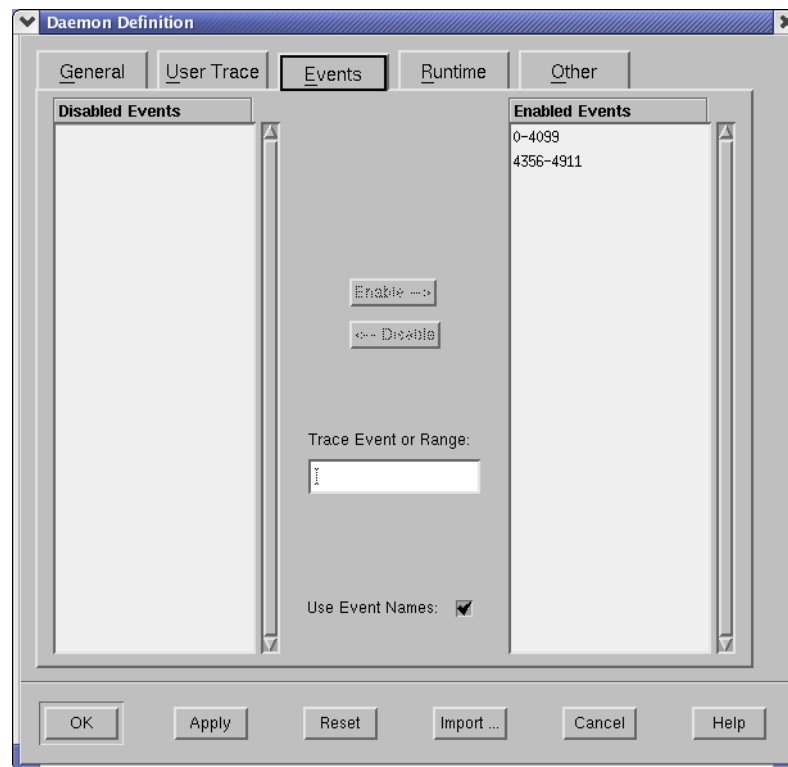
See above for details on which trace settings may be inherited.



## Events

The Events page of the Daemon Definition dialog (see “Daemon Definition Dialog” on page 5-41) allows the user to specify which trace event types will be handled by the daemon.

The user may also change this list dynamically while the daemon is executing by pressing the Trace Events... button in the Daemon Control Area of the NightTrace Main Window (see “Daemon Control Area” on page 5-32) to bring up the Enable/Disable Trace Events dialog (see “Enable / Disable Trace Events” on page 5-37).



**Figure 5-27. Daemon Definition dialog - Events**

### Disabled Events

This is a list of user trace or kernel trace event types that are disabled.

Disabled events are not logged to daemon buffers and therefore are not included in event trace outputs.

### Enabled Events

This is a list of user trace or kernel trace event types that are enabled.

Enabled events are allowed to be placed into daemon buffers and are subsequently transferred to the output device (see "Trace Events Output" on page 5-48).

**Enable -->**

Moves the selected items from the Disabled Events list or Trace Event or Range field to the Enabled Events list.

**<-- Disable**

Moves the selected items from the Enabled Events list or Trace Event or Range field to the Disabled Events list.

**Trace Event or Range**

Allows the user to enter a particular trace event type (or range of trace event types) and subsequently Enable --> or Disable --> it.

The user may use the event name associated with the event type (e.g. TR\_SYSCALL\_RESUME) or the numerical value of the trace event type (e.g. 4131).

The user may also enter a range of values either using the event names or their numerical values (e.g. TR\_INTERRUPT\_ENTRY-TR\_EXCEPTION\_EXIT or 4112-4117).

**Use Event Names**

Allows the user to view the event names of the trace event types in the Disabled Events and Enabled Events lists instead of their numerical values.

For user trace events, the user may load user-defined event map files which associate meaningful names with the user trace event ID numbers (see "Event Map Files" on page 4-10).

## Runtime

The Runtime page of the Daemon Definition dialog (see “Daemon Definition Dialog” on page 5-41) allows the user to specify the scheduling policy, CPU bias, and memory binding policies for the daemon.

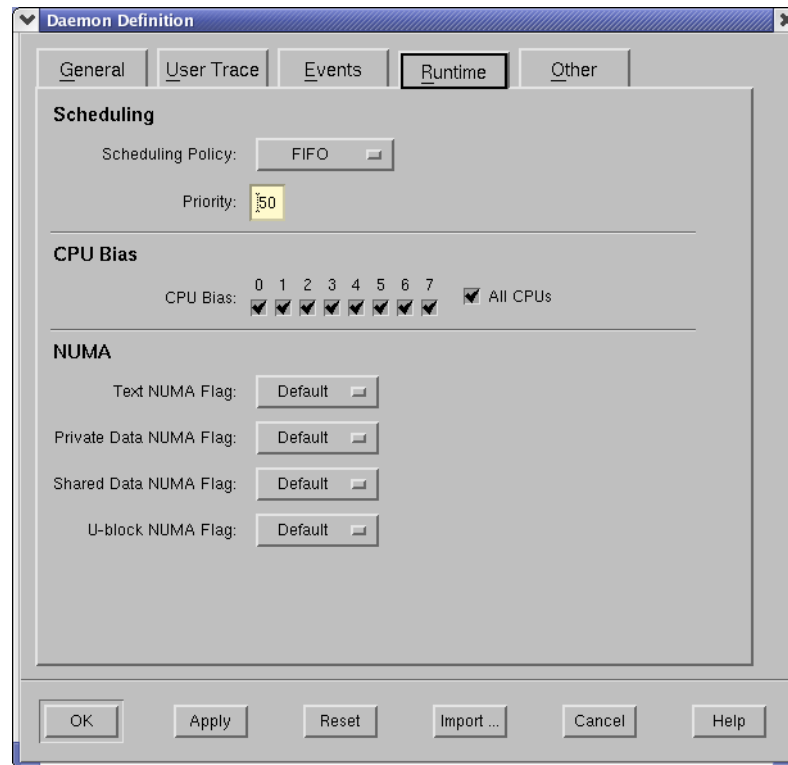


Figure 5-28. Daemon Definition dialog - Runtime

## Scheduling

### Scheduling Policy

POSIX defines three types of policies that control the way a process is scheduled by the operating system. They are `SCHED_FIFO` (**FIFO**), `SCHED_RR` (**Round Robin**), and `SCHED_OTHER` (**Time-Sharing**). Each of these scheduling policies is associated with one of the System V scheduler classes. See either the *PowerMAX OS Programming Guide* (0890423) or the *RedHawk Linux User's Guide* (0898004) for more detailed information regarding these policies and their associated classes.

### FIFO

The FIFO (first-in–first–out) policy (`SCHED_FIFO`) is associated with the fixed-priority class in which critical processes and LWPs can run in predeter-

mined sequence. Fixed priorities never change except when a user requests a change.

This policy is almost identical to the **Round Robin** (`SCHED_RR`) policy. The only difference is that a process scheduled under the **FIFO** policy does not have an associated *time quantum*. As a result, as long as a process scheduled under the **FIFO** policy is the highest priority process scheduled on a particular CPU, it will continue to execute until it voluntarily blocks.

### **Round Robin**

The **Round Robin** policy (`SCHED_RR`), like the **FIFO** policy, is associated with the fixed-priority class in which critical processes and LWPs can run in predetermined sequence. Fixed priorities never change except when a user requests a change.

A process that is scheduled under this policy (as opposed to the **FIFO** policy) has an associated time quantum.

### **Time-Sharing**

The **Time-Sharing** policy (`SCHED_OTHER`) is associated with the time-sharing class, changing priorities dynamically and assigning time slices of different lengths to processes in order to provide good response time to interactive processes and LWPs and good throughput to CPU-bound processes and LWPs.

### **Priority**

The **Priority** is relative to the selected **Scheduling Policy** (see “Scheduling Policy” on page 5-57) and the range of allowable values is dependent on the operating system.

For example, on PowerMAX OS systems, the priority values for the **FIFO** class include 0..59, where 59 is the most urgent user priority available on the system.

On RedHawk systems, the priority values for the **FIFO** class include 1..99, where 99 is the most urgent user priority available on the system.

It is recommended that a reasonable urgent priority is specified when using the **FIFO** scheduling policy to prevent event loss.

## **CPU Bias**

### **CPU Bias**

Selection of a specific CPU or set of CPUs can be advantageous to prevent event loss, reduce daemon instruction on the rest of the system, and for careful configuration to allow for relaxed Spin Lock Protection (see “Spin Lock Protection” on page 5-51).

## All CPUs

Selects all CPUs on the target system.

## NUMA

On platforms belonging to the local/global/remote subclass of non-uniform memory access (NUMA) architectures, primary memory is divided into global and local memories.

Global memory is located on a memory board where it is equally distant, in terms of access time, from all of the CPUs in the system. All CPUs share a single data path to global memory known as the system bus.

Local memory is located on a CPU board where it is closer, in terms of access time, to the co-resident CPUs. The path between a CPU and its local memory does not include the system bus. Local memory usage improves the throughput of the system in two ways: smaller access times for the co-resident CPUs and less system bus contention for the remaining CPUs.

Applications can influence the page placement decisions made by the kernel by selecting NUMA policies for different parts of their address spaces. NUMA policies specify where data should reside in the local/global/remote hierarchy.

### NOTE

These settings are ignored for non-NUMA target systems architectures, such as PowerHawk, PowerStack, and iHawk series machines.

### NOTE

These settings do not affect the memory associated with the trace buffers. Kernel trace buffers are in kernel memory allocated out of the global memory pool and user trace buffers are in shared memory allocated out of the global memory pool.

### Text NUMA Flag

This item selects the NUMA policy to apply to text (code) pages.

Text pages are those pages in private mappings that belong to a file in a file system. The traditional text segment falls into this category.

See “Policies” in the section below for a list of applicable NUMA policies.

### Private Data NUMA Flag

This item selects the NUMA policy to apply to private data pages.

Private data pages are those pages in private mappings that do not belong to a file in a file system. The traditional stack and data segments fall into this category. Note that the first time that a process writes to a page in a private, writable mapping to a file, the page will move from the text category to the private data category.

See “Policies” in the section below for a list of applicable NUMA policies.

### **Shared Data NUMA Flag**

This item selects the NUMA policy to apply to shared data pages. See “Policies” in the section below for a list of applicable NUMA policies.

### **U-block NUMA Flag**

This item selects the NUMA policy to apply to kernel data structures that contain the stack used during system calls, the register save area used during context switches, and miscellaneous other bits of information about the LWP.

See “Policies” in the section below for a list of applicable NUMA policies.

## **Policies**

Each of the above flags can be set to one of the following:

### **Global**

Specifies that the designated pages should be placed in global memory.

### **Soft Local**

Specifies that the designated pages be placed in local memory if space is available, otherwise they should be placed in global memory.

### **Hard Local**

Specifies that the designated pages must be placed in local memory.

### **Default**

Specifies that the default NUMA policy on the target system should be used.

## Other

The Other page of the Daemon Definition dialog (see “Daemon Definition Dialog” on page 5-41) allows the user to specify advanced settings with respect to the transfer of trace data from the daemon to the NightTrace display buffer.

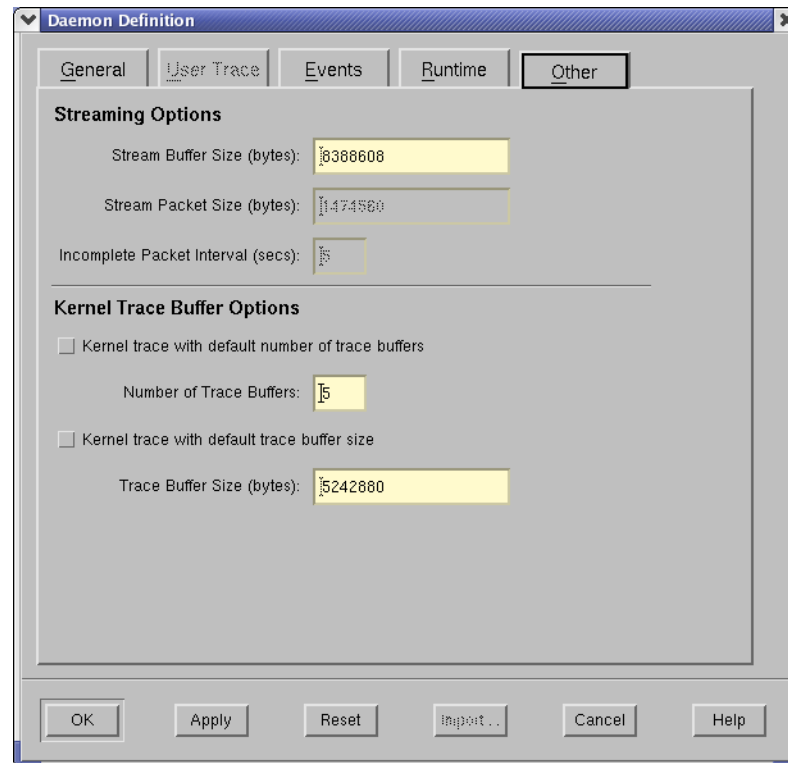


Figure 5-29. Daemon Definition dialog - Other

## Streaming Options

### Stream Buffer Size

The number of bytes for the buffer that the NightTrace uses to hold data from the daemon before sending it to the NightTrace display buffer.

#### NOTE

This is an internal buffer. You should not need to adjust the size of this buffer unless NightTrace finds that it cannot transfer data quickly enough between the daemon and the NightTrace analyzer. In such a circumstance, the daemon is forced into a Paused state (see “State” on page 5-33).

### Stream Packet Size

The amount of data (in bytes) sent from the daemon to the NightTrace analyzer for individual I/O transfers. Different network configurations may have different optimal packet sizes.

### Incomplete Packet Interval

This setting is intended for applications that have very low event rates. The user may not want to wait for a full packet (specified by the **Stream Packet Size**) to fill before the data is sent to the analyzer. If a packet cannot be filled in this amount of time, the available trace data is sent anyway.

### NOTE

The user can always hit the **Flush** button (see “Flush” on page 5-35) which causes all data in the trace buffer to be immediately transmitted across the stream.

## Kernel Trace Buffer Options

The kernel modules collect data into one or more trace buffers as events are logged. The trace daemon started by the server (**ntraceserv**) either writes events from these buffers to a file or stream.

Increasing the following settings should help avoid data loss.

### Kernel trace with default number of trace buffers

On Linux, if the default is used, the number of trace buffers used by the kernel modules to collect data defers to the server (**ntraceserv**) which starts the daemon.

#### Number of Trace Buffers

The desired number of trace buffers used by the kernel modules to collect data.

### Kernel trace with default trace buffer size

On Linux, if the default is used, the default trace buffer size defers to the server (**ntraceserv**) which starts the daemon.

#### Trace Buffer Size

The desired size of the trace buffers used by the kernel modules to collect data.



# Generating Trace Event Logs with ntraceud

---

The ntraceud Daemon .....	6-1
The Default User Daemon Configuration .....	6-2
ntraceud Modes .....	6-4
ntraceud Options .....	6-5
Option to Get Help (-help) .....	6-7
Option to Get Version Information (-version) .....	6-8
Option to Disable the IPL Register (-ipldisable) .....	6-9
Option to Prevent Page Locking (-lockdisable) .....	6-11
Option to Establish File-Wraparound Mode (-filewrap) .....	6-12
Option to Establish Buffer-Wraparound Mode (-bufferwrap) .....	6-13
Option to Define Shared Memory Buffer Size (-memsize) .....	6-16
Option to Set Timeout Interval (-timeout) .....	6-17
Option to Set the Buffer-Full Cutoff Percentage (-cutoff) .....	6-18
Option to Select Timestamp Source (-clock) .....	6-19
Option to Reset the ntraceud Daemon (-reset) .....	6-20
Option to Quit Running ntraceud (-quit) .....	6-21
Option to Present Statistical Information (-stats) .....	6-22
Option to Disable Logging (-disable) .....	6-24
Option to Enable Logging (-enable) .....	6-26
Invoking ntraceud .....	6-28



## Generating Trace Event Logs with ntraceud

A user daemon is required in order to capture trace events logged by user applications. There are two methods for controlling user daemons:

- Use the graphical user interface provided in the **ntrace** dialog as described in the preceding chapter
- Use the command line tool **ntraceud**

The interactive is often more convenient and easier to use and additionally offers concurrent viewing of trace events while the application continues to log trace data; this optional feature is called *streaming*. Alternatively, the **ntraceud** command line tool is useful in scripts where automation is required.

This chapter describes the **ntraceud** command line tool broken down into the following topics:

- The **ntraceud** daemon
- The default user daemon configuration
- **ntraceud** modes
- **ntraceud** options
- Invoking **ntraceud**

### The ntraceud Daemon

When you start up **ntraceud**, it creates a daemon background process and returns your prompt. The daemon creates a shared memory buffer in global memory. Your application writes trace events into this buffer, and the daemon copies these trace events to a trace event file.

You supply the name of the trace event file on your **ntraceud** invocation and in the `trace_begin()` library call in your application. If this file does not exist, **ntraceud** creates it; otherwise, **ntraceud** overwrites it. Unless your **umask (1)** setting overrides this default, **ntraceud** creates the file with mode 666, read and write permission to all users. If you want to maximize performance, use a trace event file that is local to the system where the **ntraceud** daemon and your application run.

A single **ntraceud** daemon may service several running applications or processes. Several **ntraceud** daemons can run simultaneously; the system identifies them by their distinctive trace event file names. The **ntraceud** daemon resides on your system under `/usr/bin/ntraceud`.

Whenever the daemon is idle, it sleeps. You can control the sleep interval with an **ntraceud** option. Logging a trace event may wake the daemon if the buffer-full cutoff percentage is exceeded or if shared memory becomes full of trace events. Flushing trace events from the shared memory buffer to disk always wakes the daemon.

## The Default User Daemon Configuration

Invoking **ntraceud** with a trace event file argument and without any options will attempt to start a user daemon with the default user daemon configuration. You can override defaults by invoking **ntraceud** with particular options. Table 6-1 summarizes these options. Later sections provide detailed descriptions of these options and operating modes.

However, if a user application has already been initiated, it may have specified a non-default configuration via the `trace_begin()` call. If the critical settings in the configuration defined by the user application differ from those specified by **ntraceud**, then **ntraceud** will fail to initialize with an appropriate diagnostic.

In the default configuration, all trace events are enabled for logging. Your application logs trace events to the shared memory buffer. By default, an architecture-specific timing source is utilized. For Intel-based machines, the Intel Time Stamp Counter (TSC register); for Night Hawk 6000 series machine, the interval timer; for PowerHawk and PowerStack series machines, the Time Base Register (TBR). However, the Real-Time Clock and Interrupt Module (RCIM) can be used as a timestamp source by using the **-clock** option to **ntraceud** (see “Option to Select Timestamp Source (-clock)” on page 6-19).

The **ntraceud** daemon operates in *expansive mode*. In expansive mode, **ntraceud** copies all trace events from the shared memory buffer to the trace event file. This behavior differs from file-wraparound mode and buffer-wraparound mode. If the trace event file does not exist when **ntraceud** starts up, **ntraceud** creates it; otherwise, **ntraceud** overwrites it.

**ntraceud** and the NightTrace library routines use page locking to prevent page faults during trace event logging.

**ntraceud** uses high-performance, low-intrusion spin locks to protect critical sections involved in logging and consuming trace events.

These spin locks require preemption control so that processes on the same CPU don't preempt a daemon or user process in the middle of a critical section and then spin forever waiting for it to be unlocked.

The spin locks are held for extremely short periods of time.

Failure to properly select a protection level may result in a process spinning forever on a CPU in the event of unfortunate preemption.

In more severe cases, the system may hang; this is only a problem if a user-level interrupt preempts another user process or daemon at an unfortunate time and attempts to log trace events to the same trace daemon session.

Data corruption of the trace data will not occur in any case.

**<default option>**

IPL protection is the safest form of preemption control as it prevents even machine interrupts from preempting the locking process. This is required when a user application will be logging events in a user-level interrupt handler (i.e. at system interrupt level).

This is the default preemption control mechanism for PowerMAX OS systems.

**NOTE**

This mechanism is not available on RedHawk systems. If selected for RedHawk systems, it is silently translated to the **-resched** protection as described below.

**-resched**

This level of protection is sufficient for user applications that log trace events as long as no user-level interrupt handlers will be logging trace events. This is the default preemption control setting for RedHawk Linux.

**-ipldisable**

Selecting no protection opens up the real possibility that the user process or the daemon could preempt each other if they are allowed to operate on the same CPU.

However, appropriate use of CPU bindings and/or priority at which the daemon operates can prevent such occurrences. Thus, if the user also takes similar care to schedule their user applications then selecting this is sufficient.

Page locking is required to prevent preemption while holding a spin lock. Without this choice, it is possible that a page fault could occur while a spin lock is held, allowing for a user application or daemon to spin forever.

When **ntraceud** is idle, it sleeps. The process of copying trace events from the shared memory buffer to a trace event file is called *flushing the buffer*. **ntraceud** wakes up and flushes the buffer when any of these conditions exist:

- **ntraceud**'s sleep interval elapses
- The buffer-full cutoff percentage is exceeded
- The shared memory buffer is full of unwritten trace events
- Your application calls `trace_flush()`, `trace_trigger()`, or `trace_end()`

A summary of NightTrace configuration defaults follows.

**Table 6-1. NightTrace Configuration Defaults**

Characteristic	Default	Modifying Option
<b>ntraceud</b> sleep interval	5 seconds	<b>-timeout</b> <i>seconds</i>
Buffer-full cutoff percentage	20% full	<b>-cutoff</b> <i>percent</i>
Shared memory buffer size	16K (16,384) trace events	<b>-memsize</b> <i>count</i>
Flush mechanism	(See above)	<b>-bufferwrap</b>
Trace event file size	Indefinite	<b>-filewrap</b> <i>bytes</i>
Trace events enabled for logging	All	<b>-disable</b> <i>ID</i> and <b>-enable</b> <i>ID</i>
Page-fault handling	Page locking	<b>-lockdisable</b>
Preemption control	Modify IPL register (PowerMAX OS)	<b>-ipldisable</b>
	Rescheduling variables (RedHawk Linux)	<b>-resched</b>

## ntraceud Modes

NightTrace can operate in three modes: expansive (default), file-wraparound, and buffer-wraparound. As the following two tables show, these modes meet different needs and have different characteristics. They differ mainly by their handling of the shared memory buffer and the trace event file on disk.

By default, NightTrace operates in expansive mode. NightTrace operates in file-wraparound mode when you specify the **-filewrap** option on the **ntraceud** invocation line. The **ntraceud -bufferwrap** option puts NightTrace in buffer-wraparound mode. See “Option to Establish File-Wraparound Mode (-filewrap)” on page 6-12 and “Option to Establish Buffer-Wraparound Mode (-bufferwrap)” on page 6-13 for more information on these options.

It is not possible to combine expansive mode with either file-wraparound or buffer-wraparound mode. Although you can mix file-wraparound and buffer-wraparound modes, it is not recommended.

Table 6-2 provides some guidelines to help you decide which mode to use.

**Table 6-2. Mode-Selection Guidelines**

Constraint	MODE		
	Expansive	File-Wraparound	Buffer-Wraparound
Trace event importance	All trace events are important	Newest trace events are important	Events just before a <code>trace_flush()</code> are important
General	Disk space and memory are plentiful	Disk space is limited	Program will run a long time

Table 6-3 shows how each NightTrace operating mode reacts to a particular condition. The process of copying trace events from the shared memory buffer to the trace event file on disk is called *flushing the buffer*.

**Table 6-3. NightTrace Operating Modes**

Condition	MODE		
	Expansive	File-Wraparound	Buffer-Wraparound
<b>ntraceud</b> sleep interval exceeded ( <b>-timeout</b> )	Flush the buffer	Flush the buffer	(No reaction)
Buffer-full cutoff percentage exceeded ( <b>-cutoff</b> )	Flush the buffer	Flush the buffer	(No reaction)
Shared memory buffer is full ( <b>-memsize</b> )	Flush the buffer	Flush the buffer	Overwrite the buffer's oldest trace events with the newest ones
Trace event file is full ( <b>-filewrap</b> )	N/A	Overwrite the file's oldest trace events with the newest ones	N/A

## ntraceud Options

**ntraceud** always copies trace events from the shared memory buffer to the trace event file, *trace\_file*. You can override some other NightTrace defaults by invoking **ntraceud**

with option(s). You can also use options to quit running or reset **ntraceud** and to obtain version, statistical, or invocation-syntax information. The full **ntraceud** invocation syntax is:

```
ntraceud [-help] [-version] [-ipldisable] [-lockdisable]
          [-filewrap bytes] [-bufferwrap] [-memsize count]
          [-timeout seconds] [-cutoff percent] [-clock source]
          [-reset] [-quit] [-quit!] [-stats] [-join]
          [[-disable ID[-ID]] [...]] [[-enable ID[-ID]] [...]] trace_file
```

You can abbreviate all **ntraceud** options to their shortest unambiguous length, but most of the examples in this manual use the long option name. These options are case-insensitive. The following examples are all equivalent:

```
ntraceud -help
ntraceud -hel
ntraceud -he
ntraceud -h
ntraceud -H
ntraceud -HE
ntraceud -Hel
ntraceud -HELP
```

You can invoke **ntraceud** more than once with different options during a single trace session; each invocation passes additional options and values to the running **ntraceud** daemon. Usually you do this to dynamically enable or disable trace events or to obtain current statistical information. Options that are available only at **ntraceud** start up are described that way.

The following sections discuss the **ntraceud** options.



## Option to Get Help (-help)

The **ntraceud -help** option displays the **ntraceud** invocation syntax on standard output.

### SYNTAX

```
ntraceud -help
```

### DESCRIPTION

The **ntraceud -help** option displays a brief help message showing the complete invocation syntax for **ntraceud**. Screen 6-1 shows an example of **-help** option output.

```
usage: ntraceud [-help] [-version] [-ipldisable] [-lockdisable]
  [-filewrap bytes] [-bufferwrap] [-memsize count] [-timeout seconds]
  [-cutoff percent] [-clock source] [-reset] [-quit] [-stats]
  [-disable ID[-ID]] [-enable ID[-ID]] trace_file

General options:
  -help           Write this message to standard output
  -version        Write the current ntraceud version stamp to standard
  output

Options for a new ntraceud daemon:
  -ipldisable     Disable use of the IPL register
  -lockdisable    Disable use of page locking
  -filewrap bytes Use file wraparound mode with max trace_file size in bytes
  -bufferwrap     Use shared memory buffer wraparound mode
  -memsize count  Set shared memory buffer size to specified event count
  -timeout seconds Set the ntraceud timeout to specified seconds
  -cutoff percent Flush events to disk at specified cutoff level
  -clock source   Specify source of event time stamps
    Valid values for source are:
      default      Use the default system clock
      rcim_tick    Use the RCIM synchronized tick clock

Options for an existing ntraceud daemon:
  -reset          Reset the ntraceud daemon and the trace_file
  -quit           Quit running ntraceud
  -stats          Write statistics (resource/environment) to standard output

Options for new and existing ntraceud daemons:
  -disable ID[-ID] Disable a specific event ID or ID range from logging
  -enable ID[-ID]  Enable a specific event ID or ID range to log

Files:
  trace_file      Holds events logged by your application and ntraceud
```

**Screen 6-1. Sample Output from the ntraceud -help Option**

## Option to Get Version Information (-version)

The **ntraceud -version** option displays the current **ntraceud** version stamp on standard output.

### SYNTAX

```
ntraceud -version
```

### DESCRIPTION

The **ntraceud -version** option displays version stamp information for this **ntraceud** daemon.

## Option to Disable the IPL Register (-ipldisable)

The **ntraceud -ipldisable** option disables the default use of the system's interrupt priority level (IPL) register by **ntraceud** and by the NightTrace library routines in your application.

### SYNTAX

```
ntraceud -ipldisable trace_file
```

### DESCRIPTION

On PowerMAX OS, by default, NightTrace modifies a shared memory region bound to the system's interrupt priority level (IPL) register to control preemption.

On RedHawk Linux, by default, NightTrace uses rescheduling variables to prevent process preemption (this does not prevent preemption by machine interrupts, but this is not of concern on RedHawk Linux since user applications cannot run at interrupt level).

On PowerMAX OS, if your application lacks read and write privilege to **/dev/sp1**, the NightTrace daemon and library initialization routine exit with errors.

On RedHawk Linux, if your application lacks privileges to be able to use rescheduling variables, the NightTrace daemon and library initialization routines will exit with errors.

If you still want to trace events, you must invoke the **ntraceud** daemon with the **-ipldisable** option. If you use the **-ipldisable** option, you must start up **ntraceud** with it.

You must not use the **-ipldisable** option if your user-level interrupt routine logs trace events to the shared memory buffer.

### CAUTION

The **-ipldisable** option should be used with great care to avoid deadlock. This may occur if more than one LWP, each biased to run on the same CPU, is logging trace events to a trace file created by an **ntraceud** invoked with the **-ipldisable** option.

Consider the following scenario: an LWP, preparing to log a trace event, locks the spin lock to protect the shared memory buffer. It is preempted by a second LWP which also attempts to log a trace event. However, due to priority inversion, the first LWP cannot release the spin lock, causing the second LWP to loop infinitely.

waiting for the spin lock to be released.

This deadlock could be avoided if **ntraceud** were invoked without the **-ipldisable** option. This would allow the first LWP to release the spin lock before being preempted.

**SEE ALSO**

For more information on the IPL register, see the *PowerMAX OS Programming Guide*.

## Option to Prevent Page Locking (-lockdisable)

The **ntraceud -lockdisable** option disables default page locking by **ntraceud** and by the NightTrace library routines in your application.

### SYNTAX

```
ntraceud -lockdisable trace_file
```

### DESCRIPTION

You can identify a running **ntraceud** daemon by its trace event file name, *trace\_file*.

By default, NightTrace locks its pages in memory. This capability prevents page faults during trace event logging that could distort trace event timings.

If you lack sufficient privileges required to lock your pages in memory, your invocation of **ntraceud** and your application exit with errors. If you still want to trace events, you must invoke the **ntraceud** daemon with the **-lockdisable** option. This option makes **ntraceud** and the NightTrace library routines in your application run without locking their pages in memory. If you use the **-lockdisable** option, you must start up **ntraceud** with it.

You must not use the **-lockdisable** option if your user-level interrupt routine logs trace events to the shared memory buffer.

## Option to Establish File-Wraparound Mode (-filewrap)

By default, the trace event file can grow indefinitely. With the **ntraceud -filewrap** option, you can make NightTrace operate in file-wraparound mode, rather than expansive mode. In file-wraparound mode, you limit the trace event file size.

### SYNTAX

```
ntraceud -filewrap bytes trace_file
```

### DESCRIPTION

The **ntraceud -filewrap** option lets you specify the maximum byte size, *bytes*, of the trace event file, *trace\_file*. Specify the *bytes* parameter as a number of bytes or as a number with a K or M suffix to show that the *bytes* parameter is in kilobyte or megabyte units, respectively. For example, 12K means 12,288 bytes. If you use the **-filewrap** option, you must start up **ntraceud** with it.

Your application logs enabled trace events into a shared memory buffer. **ntraceud** copies these trace events to the trace event file. In expansive mode, this file can grow indefinitely.

The **ntraceud -filewrap** option makes NightTrace operate in file-wraparound mode, rather than in expansive mode. In file-wraparound mode the trace event file can become full of trace events. When this happens, **ntraceud** overwrites the oldest trace events in the beginning of the file with the newest ones, intentionally discarding the oldest trace events to make room for the newest ones.

In expansive (default) and file-wraparound modes, you control automatic buffer flushing by setting the **ntraceud** sleep interval, shared memory size, and buffer-full cutoff percentage. In contrast, there is no automatic buffer flushing in buffer-wraparound mode; these values have no effect in this mode.

File-wraparound mode can be beneficial if you are short of disk space. With this mode, you specify the maximum size of the trace event file, instead of allowing it to grow indefinitely. Consider using this option if you are interested only in the most recent of many trace events logged by an application over a long period of time. If you want to determine how much disk space is available, run the **df (1)** command with the **-k** option and look at the “avail” column.

### SEE ALSO

For a comparison of expansive, file-wraparound, and buffer-wraparound modes, see “ntraceud Modes” on page 6-4.

## Option to Establish Buffer-Wraparound Mode (-bufferwrap)

The process of copying trace events from the shared memory buffer to the trace event file on disk is called *flushing the buffer*. With the **ntraceud -bufferwrap** option, you can make NightTrace operate in buffer-wraparound mode, rather than expansive mode. In buffer-wraparound mode, the **ntraceud** daemon flushes only the most recent trace events, rather than all trace events. Your application asynchronously triggers every buffer flush.

### SYNTAX

```
ntraceud -bufferwrap trace_file
```

### DESCRIPTION

The **ntraceud** daemon always logs enabled trace events into a shared memory buffer. In expansive mode, when the buffer is full (or when some other conditions exist), **ntraceud** automatically flushes the buffer to the trace event file, *trace\_file*.

The **ntraceud -bufferwrap** option makes NightTrace operate in buffer-wraparound mode, rather than in expansive mode. When the buffer is full in buffer-wraparound mode, the application treats the shared memory buffer as a circular queue and overwrites the oldest trace events with the newest ones, intentionally discarding the oldest trace events to make room for the newest ones. This overwriting continues until your application explicitly calls `trace_flush()`. Only then, does **ntraceud** copy the remaining trace events from the shared memory buffer to the trace event file. If you use the **-bufferwrap** option, you must start up **ntraceud** with it.

### NOTE

You control automatic buffer flushing by setting the **ntraceud** sleep interval and buffer-full cutoff percentage in expansive (default) mode and in file-wraparound mode. In contrast, there is no automatic buffer flushing in buffer-wraparound mode; these values have no effect in this mode. Invoking **ntraceud** with the **-bufferwrap** option, makes **ntraceud** ignore any **-time-out** and **-cutoff** options.

In buffer-wraparound mode, you can estimate the maximum number of trace events to be written to your trace event file by using the following formula:

$$\text{max\_events} = \text{max\_events\_in\_buffer} * \text{flush\_count}$$

where:

`max_events`      The maximum number of trace events.

<code>max_events_in_buffer</code>	The number of trace events the shared memory buffer can hold. You can set this value when you invoke <b>ntraceud</b> with the <b>-memsize</b> option.
<code>flush_count</code>	The number of <code>trace_flush()</code> calls your application executes.

For example, if you set your shared memory buffer size to 1000 trace events, then `max_events_in_buffer` is 1000. If you expect your three `trace_flush()` calls to execute two times each, then `flush_count` is six (3 \* 2). Calculating `max_events` gives you about 6000 (1000 \* 6) trace events in your trace event file.

Buffer-wraparound mode:

- Can help you with debugging
- Can reduce trace events to a manageable number
- May conserve disk space

Buffer-wraparound mode can be useful in debugging.

Assume that you are debugging a fault in a large application. Follow the steps below to accomplish your task.

1. Insert a `trace_flush()` call in your code where you believe the fault occurs.
2. Compile and link your application.
3. Invoke **ntraceud** with the **-bufferwrap** option.
4. Run your application.

When your application executes the `trace_flush()` call, **ntraceud** copies all trace events still in the shared memory buffer to the trace event file. You can then use the **ntrace** display utility to graphically analyze only the trace events immediately preceding the fault.

Buffer-wraparound mode can also be useful in reducing trace events to a manageable number. In this mode, when the shared memory buffer is full, the newest trace events overwrite the oldest ones. This means that if the shared memory buffer becomes full before your application executes the `trace_flush()` call, **ntraceud** copies only the current contents of the buffer to the trace event file. This way, you can exclude the oldest trace events from your **ntrace** displays.

In buffer-wraparound mode, **ntraceud** usually flushes fewer trace events to the trace event file than in expansive mode. Thus, this mode can conserve disk space.

If you want to determine how much disk space is available, run the **df (1)** command with the **-k** option and look at the “avail” column. Use the following command to see the system settings for the current, default, minimum, and maximum shared memory segment size:

```
$ /etc/conf/bin/idtune -g SHMMAX
```

See the **idtune (1M)** man page for more information.



**SEE ALSO**

For more information on `trace_flush()`, see “`trace_flush()` and `trace_trigger()`” on page 2-21. For a comparison of expansive, file-wraparound, and buffer-wraparound modes, see “`ntraceud` Modes” on page 6-4. For information on limiting the number of logged trace events, see “Option to Define Shared Memory Buffer Size (`-memsize`)” on page 6-16.

## Option to Define Shared Memory Buffer Size (-memsize)

By default, the shared memory buffer can hold 16,384 trace events. When the buffer is full of unwritten trace events, the **ntraceud** daemon wakes up and copies the trace events to the trace event file. The **ntraceud -memsize** option lets you alter the size of the shared memory buffer.

### SYNTAX

```
ntraceud -memsize count trace_file
```

### DESCRIPTION

The **ntraceud -memsize** option lets you set the shared memory buffer size. Specify the *count* parameter as a maximum number of trace events or as a number with a K or M suffix to show that the *count* parameter is in kilobyte or megabyte units, respectively. For example, 12K means 12,288 trace events. **ntraceud** rounds that number up to a full page boundary. A trace event with zero or one argument takes up 16 bytes; a trace event with more arguments takes up 32 bytes: 16 bytes for the basic trace event and one argument and 16 bytes for the NT\_CONTINUE overhead trace event and the remaining arguments.

On PowerMAX OS, use the following command to see the system settings for the current, default, minimum, and maximum shared memory segment size:

```
$ /etc/conf/bin/idtune -g SHMMAX
```

See the **idtune (1M)** man page for more information.

By default, if the shared memory buffer becomes full, **ntraceud** wakes up and copies trace events from the shared memory buffer to the trace event file, *trace\_file*. You can increase the *count* parameter to prevent trace event loss. If you use the **-memsize** option, you must start up **ntraceud** with it.

By changing the shared memory buffer size, you can:

- Alter the buffer flush frequency
- Control the number of trace events copied to the trace event file in buffer-wraparound mode

### SEE ALSO

For information on limiting the number of logged trace events, see “Option to Establish Buffer-Wraparound Mode (-bufferwrap)” on page 6-13.

## Option to Set Timeout Interval (-timeout)

By default, **ntraceud** sleeps 5 seconds after writing trace events to disk. The **ntraceud -timeout** option lets you set this timeout interval.

### SYNTAX

```
ntraceud -timeout seconds trace_file
```

### DESCRIPTION

You can identify a running **ntraceud** daemon by its trace event file name, *trace\_file*.

When **ntraceud** is idle, the daemon sleeps. By default, the sleep interval is a maximum of 5 seconds. The **ntraceud -timeout** option lets you establish the maximum number of seconds, *seconds*, that the **ntraceud** daemon sleeps.

Waking the **ntraceud** daemon incurs overhead that can distort trace event timings; decreasing the timeout parameter makes it more likely that the daemon will be awake when needed. You can also decrease the timeout parameter to prevent trace event loss. Note: If your application does not log events frequently, you can increase the timeout to reduce the time the daemon runs and consumes CPU cycles.

If you use the **-timeout** option, you must start up **ntraceud** with it. If you invoke **ntraceud** with both the **-timeout** and **-bufferwrap** options, **ntraceud** ignores the **-timeout** option.

**ntraceud** does not sleep for the full period if:

- Your application executes a call to `trace_flush()`, `trace_trigger()`, or `trace_end()`
- Your application logs a trace event that causes shared memory to become full or your buffer-full cutoff percentage to be reached
- You specify a timeout parameter which exceeds the time in which the lower 32 bits of the timestamp source would roll over. This rollover time varies from architecture to architecture (with a minimum value of 257.69803 seconds) and is calculated by **ntraceud** as part of its initialization. It is important to detect this rollover so that proper ordering of trace events is maintained. If you specify a timeout interval which exceeds the rollover time, **ntraceud** uses the rollover time as the timeout interval, ignoring the value specified.

## Option to Set the Buffer-Full Cutoff Percentage (-cutoff)

By default, when the shared memory buffer becomes 20-percent full of unwritten trace events, the **ntraceud** daemon wakes up and copies the trace events to the trace event file. The **ntraceud -cutoff** option lets you alter this percentage.

### SYNTAX

```
ntraceud -cutoff percent trace_file
```

### DESCRIPTION

The **ntraceud -cutoff** option lets you set the buffer-full cutoff percentage, *percent*, for the shared memory buffer. *percent* is an integer percentage in the range 0-99, inclusive.

The process of copying trace events from the shared memory buffer to the trace event file, *trace\_file*, on disk is called *flushing the buffer*. When a logged trace event causes the buffer to reach the buffer-full cutoff percentage, **ntraceud** wakes up and flushes the buffer.

Waking the **ntraceud** daemon incurs overhead that can distort trace event timings; decreasing the shared memory buffer-full cutoff percentage makes it more likely that the daemon will be wakened by the application. You can also decrease the *percent* parameter to prevent trace event loss; the effect is an increase in the buffer flush frequency.

If you use the **-cutoff** option, you must start up **ntraceud** with it. If you invoke **ntraceud** with both the **-cutoff** and **-bufferwrap** options, **ntraceud** ignores the **-cutoff** option.

## Option to Select Timestamp Source (-clock)

The **ntraceud -clock** option allows you to select which timing source will be used to timestamp events.

### SYNTAX

```
ntraceud -clock source trace_file
```

### DESCRIPTION

The **ntraceud -clock** option lets you select the timing source used to timestamp trace events. Valid *source* values are:

<b>default</b>	the interval timer (NightHawk 6000 Series) or the Time Base Register (Power Hawk/PowerStack)
<b>rcim_tick</b>	the RCIM synchronized tick clock

If you invoke **ntraceud** with the **-clock** option, you must supply a value for the *source*.

If **rcim\_tick** is specified for the *source* and the system on which you are tracing does not have an RCIM installed or configured or if the RCIM synchronized tick clock on the system on which you are tracing is stopped, the NightTrace daemon and library initialization routine exit with errors.

If the **-clock** option is not specified, the interval timer (NightHawk 6000 Series) or the Time Base Register (Power Hawk/PowerStack) is used to timestamp trace events.

## Option to Reset the ntraceud Daemon (-reset)

The **ntraceud -reset** option resets a running **ntraceud** daemon process.

### SYNTAX

```
ntraceud -reset trace_file
```

### DESCRIPTION

Running **ntraceud** daemons are located using the shared memory identifier keyed by the trace event file name, *trace\_file*.

By default, **ntraceud** overwrites the trace event file if it is not currently in use. In contrast, the **ntraceud -reset** option empties the file and prepares the running daemon for another trace run. Use the **-reset** option when you are no longer interested in the contents of an active trace event file. You can invoke **ntraceud** multiple times with the **-reset** option.

### SEE ALSO

For information on quitting an **ntraceud** session without clearing the trace event file, see “Option to Quit Running **ntraceud** (-quit)” on page 6-21.

## Option to Quit Running `ntraceud` (-quit)

The `ntraceud -quit` and `-quit!` options terminate a running `ntraceud` process.

### SYNTAX

```
ntraceud -quit trace_file
ntraceud -quit! trace_file
```

### DESCRIPTION

Running `ntraceud` daemons are located using the shared memory identifier keyed by the trace event file name, `trace_file`.

A process completes its NightTrace session by calling `trace_end()` or exiting normally. The `-quit` and `-quit!` option instruct `ntraceud` to terminate tracing. When `-quit` is used, `ntraceud` will wait for all user processes associated with this daemon that are currently tracing to terminate, whereas use of `-quit!` skips this check. The following actions are then taken:

- Remaining trace events are flushed to the trace event file
- The output file is closed
- The shared memory buffer is removed (unless user applications still exist)
- The running `ntraceud` daemon terminates

#### TIP:

You cannot get statistical information after you quit running `ntraceud`. Consider getting statistical information before you quit running `ntraceud`. For statistical information on your trace session, see “Option to Present Statistical Information (-stats)” on page 6-22.

Assume that you have invoked `ntraceud` with the `-quit` option, and you want to reinvoke `ntraceud` with the same trace event file. Your next `ntraceud` invocation will automatically overwrite the trace event file.

### SEE ALSO

For information on resetting `ntraceud` and the trace event file for another session, see “Option to Reset the `ntraceud` Daemon (-reset)” on page 6-20.

## Option to Present Statistical Information (-stats)

The **ntraceud -stats** option presents a display of statistical information for a running **ntraceud** daemon on standard output.

### SYNTAX

```
ntraceud -stats trace_file
```

### DESCRIPTION

Running **ntraceud** daemons are located using the shared memory identifier keyed by the trace event file name, *trace\_file*.

The **-stats** option provides statistical information that tells you about your current NightTrace configuration and resource use. This information can help you determine if you have adequate resources for your application. If you are interested in watching changes in the statistics, invoke **ntraceud** multiple times with the **-stats** option.

Specifically, the **-stats** option provides information on:

- **ntraceud** mode. **ntraceud** may run in the following modes:
  - **NT\_M\_DEFAULT**, meaning expansive (default) mode
  - **NT\_M\_FILEWRAP**, meaning file-wraparound mode
  - **NT\_M\_BUFFERWRAP**, meaning buffer-wraparound mode
- Shared memory buffer size
- Buffer-full cutoff percentage
- **ntraceud** timeout interval
- Number of threads or processes logging in your application
- Number of times trace events were lost. This statistic refers to a situation that infrequently arises during a NightTrace session. **ntraceud** may lose some trace events if the trace events enter the shared memory buffer faster than **ntraceud** can copy them to the trace event file. For more information on this topic, see “Preventing Trace Events Loss” on page A-1.
- Number of automatic buffer flushes (For more information on buffer flushes, see “`trace_flush()` and `trace_trigger()`” on page 2-21.)
- Number of trace events logged to shared memory. **ntraceud** and some NightTrace library routines occasionally log predefined trace events into the shared memory buffer. Therefore, the statistic for number of trace events logged to shared memory may exceed the number of times your application logs a trace event.
- Trace event IDs enabled

Screen 6-2 shows a sample of **-stats** option output.



```
$ ntraceud -stats log

NTRACEUD STATISTICS

The ntraceud daemon is running in NT_M_DEFAULT mode.
There is a maximum of 16384 trace events in the shared memory buffer
The buffer-full threshold is 20% or 3276 trace events
The daemon timeout period is 5 seconds
There are 1 thread(s) logging trace events
The shared memory buffer had 0 events lost
There have been 0 unrequested buffer flushes
The total number of trace events logged to shared memory is 5

Enabled Events:
0-4095
```

### Screen 6-2. Sample Output from ntraceud -stats Option

Defaults for some of these values exist in the header file `/usr/include/ntrace.h`. You can override the default values with `ntraceud` options. See Table 6-1 for more information on the default values and the corresponding options used to override them.

### SEE ALSO

For information on trace event loss prevention, see “Option to Establish File-Wrap-around Mode (-filewrap)” on page 6-12, “Option to Set Timeout Interval (-timeout)” on page 6-17, and “Option to Set the Buffer-Full Cutoff Percentage (-cutoff)” on page 6-18.

## Option to Disable Logging (-disable)

By default, all trace events are enabled for logging to the shared memory buffer. The **ntraceud -disable** option makes the application ignore requests to log a specific trace event or range of trace events.

### SYNTAX

```
ntraceud -disable ID [...] trace_file  
ntraceud -disable ID_low-ID_high [...] trace_file
```

### DESCRIPTION

Sometimes **ntraceud** logs so many trace events that it is hard to understand the **ntrace** display. Occasionally you know that a particular trace event or trace event range is not interesting at certain times but is interesting at others. When either of these conditions exist, it is useful to disable the extraneous trace events. You can disable trace events temporarily, where you disable and later re-enable them. You can also disable trace events permanently, where you disable them before the application runs or during its execution and never re-enable them.

In the first format, the **ntraceud -disable** option dynamically disables a specific trace event ID, *ID*, from logging to the shared memory buffer. In the second format, the **ntraceud -disable** option dynamically disables a range of trace event IDs, *ID\_low* through *ID\_high*, from logging to the shared memory buffer. In either case, trace event IDs are integers in the range 0-4095, inclusive. At defined times, **ntraceud** copies trace events from the shared memory buffer to the trace event file, *trace\_file*.

### NOTE

The **-disable** option disables trace events in all processes that rely on the same **ntraceud** daemon to log to the same trace event file.

This first format provides the same functionality as the `trace_disable()` NightTrace library routine. The second format provides the same functionality as the `trace_disable_range()` NightTrace library routine. One advantage of using the **-disable** option rather than the library routine is that you do not have to re-edit, recompile, and relink your application. For more information on disable library routines, see “`trace_enable()`, `trace_disable()`, and Their Variants” on page 2-17.

Note: In the following text, the names of the trace event files are varied for interest.

You can start up **ntraceud** with the **-disable (-d)** option. You can also re-invoke **ntraceud** with this option while **ntraceud** is running. Furthermore, using the **-disable** option to disable an already disabled trace event has no effect. For example, assume that you invoke **ntraceud** three times, sequentially, before your application terminates and that **ntraceud** has not logged to the `ntoutput` file before.

```
$ ntraceud -d4 ntoutput -- trace event 4 is disabled
$ ntraceud -d7 ntoutput -- trace events 4 & 7 are now disabled
$ ntraceud -d4 ntoutput -- no effect; trace events 4 & 7 disabled
```

There may be any number of **-disable** options on an **ntraceud** invocation line. The following example illustrates this fact.

```
$ ntraceud -d10 -d15 mytrace -- trace events 10 & 15 are disabled
```

You may specify a hyphenated trace event range on the **ntraceud** invocation line. The following example depicts this case.

```
$ ntraceud -d23-25 traceoutput -- events 23, 24, and 25 disabled
```

The following two sequences show how important timing can be when you use the **-disable** option. The same steps appear in both sequences, but their order differs. When the first sequence ends, nothing has been logged and all trace events are enabled. In contrast, when the second sequence ends, trace event 52 has been logged once and is now disabled.

**Table 6-4. ntraceud Disable Sequence #1**

From the Shell	From the Application	Comments
1. Invoke <b>ntraceud</b>		All trace events enabled
2. Invoke <b>ntraceud -d52</b>		Trace event 52 disabled
3. Start application		
4.	Call <code>trace_event(52)</code>	Trace event 52 <u>not</u> logged
5.	Call <code>trace_enable(52)</code>	Trace event 52 enabled

**Table 6-5. ntraceud Disable Sequence #2**

From the Shell	From the Application	Comments
1. Invoke <b>ntraceud</b>		All trace events enabled
2. Start application		Trace event 52 enabled
3.	Call <code>trace_event(52)</code>	Trace event 52 logged
4.	Call <code>trace_enable(52)</code>	No effect
5. Invoke <b>ntraceud -d52</b>		Trace event 52 disabled

**SEE ALSO**

For information on enabling trace events, see “Option to Enable Logging (-enable)” on page 6-26 and “`trace_enable()`, `trace_disable()`, and Their Variants” on page 2-17.

## Option to Enable Logging (-enable)

By default, all trace events are enabled for logging to the shared memory buffer. The **ntraceud -enable** option makes the application notice previously disabled requests to log a specific trace event or a range of trace events.

### SYNTAX

```
ntraceud -enable ID [...] trace_file
ntraceud -enable ID_low-ID_high [...] trace_file
```

### DESCRIPTION

In the first format, the **ntraceud -enable** option dynamically re-enables a specific disabled trace event ID, *ID*, for logging to the shared memory buffer. In the second format, the **ntraceud -enable** option dynamically re-enables a range of disabled trace event IDs, *ID\_low* through *ID\_high*, for logging to the shared memory buffer. In either case, trace event IDs are integers in the range 0-4095, inclusive. At defined times, **ntraceud** copies trace events from the shared memory buffer to the trace event file, *trace\_file*.

### NOTE

The **-enable** option affects all processes that rely on the same **ntraceud** daemon to log to the same trace event file.

The first format provides the same functionality as the `trace_enable()` NightTrace library routine. The second format provides the same functionality as the `trace_enable_range()` NightTrace library routine. One advantage of using the **ntraceud** option instead of the library routine is that you do not have to re-edit, recompile, and relink your application. For more information on enable library routines, see “`trace_enable()`, `trace_disable()`, and Their Variants” on page 2-17.

In the following text, the names of the trace event files are varied for interest. Unless otherwise stated, all the following examples describe the results of a non-startup **ntraceud** invocation.

There may be any number of **-enable (-e)** options on an **ntraceud** invocation line. The following example illustrates this fact.

```
$ ntraceud -e10 -e15 mytrace -- trace events 10 and 15 enabled
```

You may specify a hyphenated trace event range on the **ntraceud** invocation line. The following example depicts this case.

```
$ ntraceud -e23-25 traceoutput -- trace events 23, 24, & 25
                                     enabled
```

The **-enable** option acts differently when you use it:

- On **ntraceud** start up
- On later **ntraceud** invocations

If you start up **ntraceud** with the **-enable** option, the specified trace event(s) are the only one(s) enabled; all other trace events are disabled. For example, if the following invocation starts up **ntraceud**, then only trace event 18 is enabled.

```
$ ntraceud -e18 traceout
```

When you use the **-enable** option on non-startup **ntraceud** invocations, Night-Trace adds the specified trace event(s) to the list of enabled trace events. Furthermore, attempting to enable an already enabled trace event has no effect. For example, assume that you invoke **ntraceud** four times, sequentially, before your application terminates and that **ntraceud** has not logged to the `ntoutput` file before.

```
$ ntraceud ntoutput           -- all trace events enabled
$ ntraceud -d4 -d7 ntoutput -- all except 4 and 7 enabled
$ ntraceud -e4 ntoutput      -- all except 7 enabled
$ ntraceud -e4 ntoutput      -- no effect; all except 7 enabled
```

The following two sequences show how important timing can be when you use the **-enable** option. The same steps appear in both sequences, but their order differs. When the first sequence ends, nothing has been logged and all trace events are enabled. In contrast, when the second sequence ends, trace event 52 has been logged once and is now disabled.

**Table 6-6. ntraceud Enable Sequence #1**

From the Shell	From the Application	Comments
1. Invoke <b>ntraceud</b>		All trace events enabled
2. Start application		
3.	Call <code>trace_disable(52)</code>	Trace event 52 disabled
4.	Call <code>trace_event(52)</code>	Trace event 52 <u>not</u> logged
5. Invoke <b>ntraceud -e52</b>		Trace event 52 enabled

**Table 6-7. ntraceud Enable Sequence #2**

From the Shell	From the Application	Comments
1. Invoke <b>ntraceud</b>		All trace events enabled
2. Start application		
3.	Call <code>trace_event(52)</code>	Trace event 52 logged
4. Invoke <b>ntraceud -e52</b>		No effect
5.	Call <code>trace_disable(52)</code>	Trace event 52 disabled

## SEE ALSO

For information on disabling trace events, see “Option to Disable Logging (-disable)” on page 6-24 and “trace\_enable(), trace\_disable(), and Their Variants” on page 2-17.

## Invoking ntraceud

This section shows a few common **ntraceud** invocation examples. In each example, the *trace\_file* argument corresponds to the trace event file name you supply on your call to the `trace_begin()` library routine.

Normally, your first **ntraceud** invocation looks something like the following sample.

```
ntraceud trace_file
```

The next sample invocation assumes that you lack both page lock privilege (**-lockdisable**) and read and write access to `/dev/sp1` needed to modify the interrupt priority level register (**-ipldisable**), or lack sufficient privileges required for rescheduling variables.

```
ntraceud -lockdisable -ipldisable trace_file
```

The following invocation might be used when tuning your NightTrace configuration because you lost trace events last time.

```
ntraceud -memsize count -cutoff percent trace_file
```

There are several times when you may want to use the following invocation. Usually this invocation is appropriate if you are using `trace_flush()` calls to debug a fault in your application or to reduce the number of logged trace events so the **ntrace** display is more readable.

```
ntraceud -bufferwrap trace_file
```

The following invocation is also useful on several occasions. One example is if you want to conserve disk space.

```
ntraceud -filewrap bytes trace_file
```

The following invocation waits for all user applications associated with the running **ntraceud** daemon to terminate, flushes remaining trace events to the trace event file, closes the file, removes the shared memory buffer, then terminates the running **ntraceud**.

```
ntraceud -quit trace_file
```

Similarly, the following invocation immediately flushes remaining trace events to the trace file, closes the file, and terminates the running **ntraceud** daemon. User applications can continue to run and make NightTrace API calls, but no trace events will be logged. Subsequently, a new user daemon can be initiated and trace events will start being logged again:

```
ntraceud -quit! trace_file
```

At this point, you can begin data analysis.





# Generating Trace Event Logs with ntracekd

---

The ntracekd Daemon .....	7-1
ntracekd Modes .....	7-1
ntracekd Options .....	7-2
ntracekd Invocations .....	7-4



# Generating Trace Event Logs with ntracekd

A kernel daemon is required in order to capture trace events logged by the operating system kernel. There are two methods for controlling kernel daemons:

- Using the graphical user interface provided in NightTrace Main Window
- Using the command line tool **ntracekd**

The interactive method is often more convenient and easier to use and additionally offers concurrent viewing of trace events while the kernel continues to log trace data; this optional feature is called *streaming*. Alternatively, the **ntracekd** command line tool is useful in scripts where automation is required.

This chapter describes the **ntracekd** command line tool and consists of the following topics:

- The **ntracekd** daemon
- **ntracekd** modes
- **ntracekd** options
- Example **ntracekd** invocations

## The ntracekd Daemon

When you initiate **ntracekd**, it creates a daemon background process and returns while that daemon process executes. Once it returns to your prompt, the background process has already initiated kernel tracing.

You supply the name of the trace event output file on your **ntracekd** invocation. Since the capture of kernel data can quickly consume vast quantities of disk space, the **ntracekd** tool requires that you specify a limit on the size of the output file. Once the limit is reached, older kernel data in the file will be overwritten with newer data. The interface does allow you to specify an unlimited file size; however, this is not recommended.

The **ntracekd** daemon resides on your system under `/usr/bin/ntracekd`.

## ntracekd Modes

**ntracekd** essentially always operates in a file-wraparound mode, since it requires you to put a limit on the maximum size of the output file. If the limit is reached, then kernel trac-

ing continues, but newer kernel events overwrite older events in the file. When viewed by the NightTrace analyzer, the events will be appropriately displayed in chronological order.

**ntracekd** also offers a buffer-wraparound mode. This mode stipulates that the kernel continues to log kernel events to its internal buffers located in kernel memory, overwriting the oldest kernel trace events with the newest ones. No disk activity occurs until **ntracekd** is terminated, at which time, all kernel trace buffers are copied to the output file.

## ntracekd Options

The full **ntracekd** invocation syntax is:

```
ntracekd [options] filename
```

The *filename* parameter is required for all **ntracekd** invocations. When starting a daemon, it defines the output file. When requesting statistics for a running daemon or when stopping a daemon, it identifies the running daemon.

The command-line options to **ntracekd** are:

```
-b  
--bufferwrap
```

Collect events in kernel bufferwrap mode, delaying output to *filename* until stopped. This delays the disk activity normally involved in copying kernel buffers to the output file as they become full, until after kernel tracing has been stopped.

```
-q  
--quit
```

Stop an existing kernel daemon. Once kernel tracing has been stopped, all remaining trace events already logged in the kernel buffers are copied to the output file. The **ntracekd** command will not return until the copy is complete.

```
-w seconds  
--wait=seconds
```

Start the daemon and begin kernel tracing for *seconds* before stopping the daemon.

```
-r  
--rcim
```

Use the RCIM tick clock as the timing source instead of the default timing source.

**-x**  
**--raw**

Disable automatic filtration of the kernel data leaving the format of the output file as a raw kernel file. Raw kernel files can be passed directly to NightTrace which will execute the filtration process on the fly. By default, **ntracekd** filters the raw data to avoid otherwise unnecessary repetitive filtration by NightTrace. The vectors file information obtained during filtration can be retrieved from a filtered or raw file when loaded in NightTrace; it is available in the Session Overview area of the NightTrace Main Window.

**-i**  
**--info**

This option can be specified to obtain statistics about a kernel daemon already initiated by a previous **ntracekd** command. It prints statistics to **stdout**.

**-H**  
**--help**

Print a description of the available options and stop.

**-k**  
**--kill**

Kill any active kernel daemon without regard to proper shutdown procedures. This will allow subsequent kernel daemons to be initiated but data from the previous daemon may be lost.

**-s size**  
**--size=size**

This option is required when initiating a daemon and specifies the maximum size of the output file. *size* may be specified as an integer number optionally followed by a **K**, **M**, or **G** which indicates kilobytes, megabytes, or gigabytes, respectively. If no letter is specified, the units are assumed to be in bytes. *size* may also be **+**, which indicates that the output may grow without limit. Use of **+** is not recommended as kernel tracing can quickly consume vast quantities of disk space.

**-Bs sz**  
**--bufsize=sz**

This option defines the size of each kernel buffer. *sz* may be specified as an integer number optionally followed by a **K**, **M**, or **G** which indicates kilobytes, megabytes, or gigabytes, respectively. If no letter is specified, the units are assumed to be in bytes. On RedHawk systems, the default size of a kernel buffer is 250000 bytes. This option is ignored on PowerMAX OS systems.

**-Bn n**  
**--numbufs=n**

This option defines the number of kernel buffers. *n* must be a integer number. On RedHawk systems, the number of kernel buffers defaults to 4. This option is ignored on PowerMAX OS systems.

## ntracekd Invocations

A typical invocation of **ntracekd** to initiate kernel tracing would be

```
> ntracekd --size=10M kernel-data
```

This starts a kernel trace daemon in the background and specifies a maximum size limit for the output file **kernel-data** of 10 megabytes. The command returns as soon as kernel trace has begun.

To check on the status of the running daemon, the following command might be used:

```
> ntracekd --info kernel-data
status:           running
events lost:      0
events captured: 13465
events written:   13465
events in buffer: 1493
```

To terminate the running daemon, the following command would be used:

```
> ntracekd --quit kernel-data
```

To initiate a daemon to capture kernel data while a user application executes, then to terminate the daemon and view the data, the following sequence of commands might be used:

```
> ntracekd --size=10M kernel-data
> ./a.out
> ntracekd --quit kernel-data
> ntrace kernel-data
```

# Viewing Trace Event Logs



Mouse Button Operations .....	8-2
Viewing Strategy .....	8-2
Editing Single Fields .....	8-4
Editing Multiple Fields.....	8-4





## Viewing Trace Event Logs

---

NightTrace's display page has two modes: *edit mode* and *view mode*. This chapter discusses *view mode*, the mode that displays trace events and states from your trace event file(s).

NightTrace displays this information:

- graphically in configured display object(s) on the grid
- statistically in fields of the interval control area
- uniformly on all display page(s) - changes on one page are reflected on all pages

NightTrace uses the same display page(s) in both *edit mode* and *view mode*. However, toggling between modes changes the interval scroll bar, fields in the interval control area, and the push buttons. In *view mode*, the message display area shows some statistics, as well as errors and warnings. The default mode for an existing display is *view mode*. (See "Mode Button" on page 9-36 for more information.)

View mode lets you locate interesting parts of your trace session by:

- shifting with the interval scroll bar
- clicking on some of the interval push buttons
- editing some field(s) in the interval control area
- using the built-in Search tool (See Chapter 12 for more information.)

See Chapter 9 for more information on the components of the display page and Chapter 10 for detailed information about the various display objects.

## Mouse Button Operations

Mouse button operations in View mode appear in Table 8-1 and in the *NightTrace Pocket Reference* card. Unfamiliar terminology is defined later in this chapter.

**Table 8-1. View-Mode Mouse Button Operations**

Button	Use Within a Column
Mouse button 1	Move the current time line to the place where the pointer rests, or put the text cursor where you clicked in the text field.
Hold down <Ctrl> and click mouse button 1	Move the mark and the current time line to the place where the pointer rests.
Hold down <Ctrl>, hold down mouse button 1, and drag horizontally	Move the mark to the beginning point of the drag region, and move the current time line to the ending point of the drag region. The drag region is highlighted as you drag the pointer.
Mouse button 2	Write a statistic in the message display area that tells about the trace event where the pointer rests in a State Graph or Event Graph.
Hold down <Ctrl> and click mouse button 2	Write a statistic in the message display area that tells how far the pointer is from the mark. A positive number means the pointer is to the right of the mark. A negative number means the pointer is to the left of the mark.
Mouse button 3	Write a statistic in the message display area that tells about the data item where the pointer rests in a Data Graph.
	On a state graph event marker, allows you to name a tag for that event.
Hold down <Ctrl> and click mouse button 3	Write a statistic in the message display area that tells how far the pointer is from the current time line. A positive number means the pointer is to the right of the current time line. A negative number means the pointer is to the left of the current time line.

## Viewing Strategy

NightTrace is a flexible tool. Depending on your personal preferences and how much you know about your trace events, there are several ways to locate intervals of interest. The following flowchart provides information to help you decide what to do next in View mode.

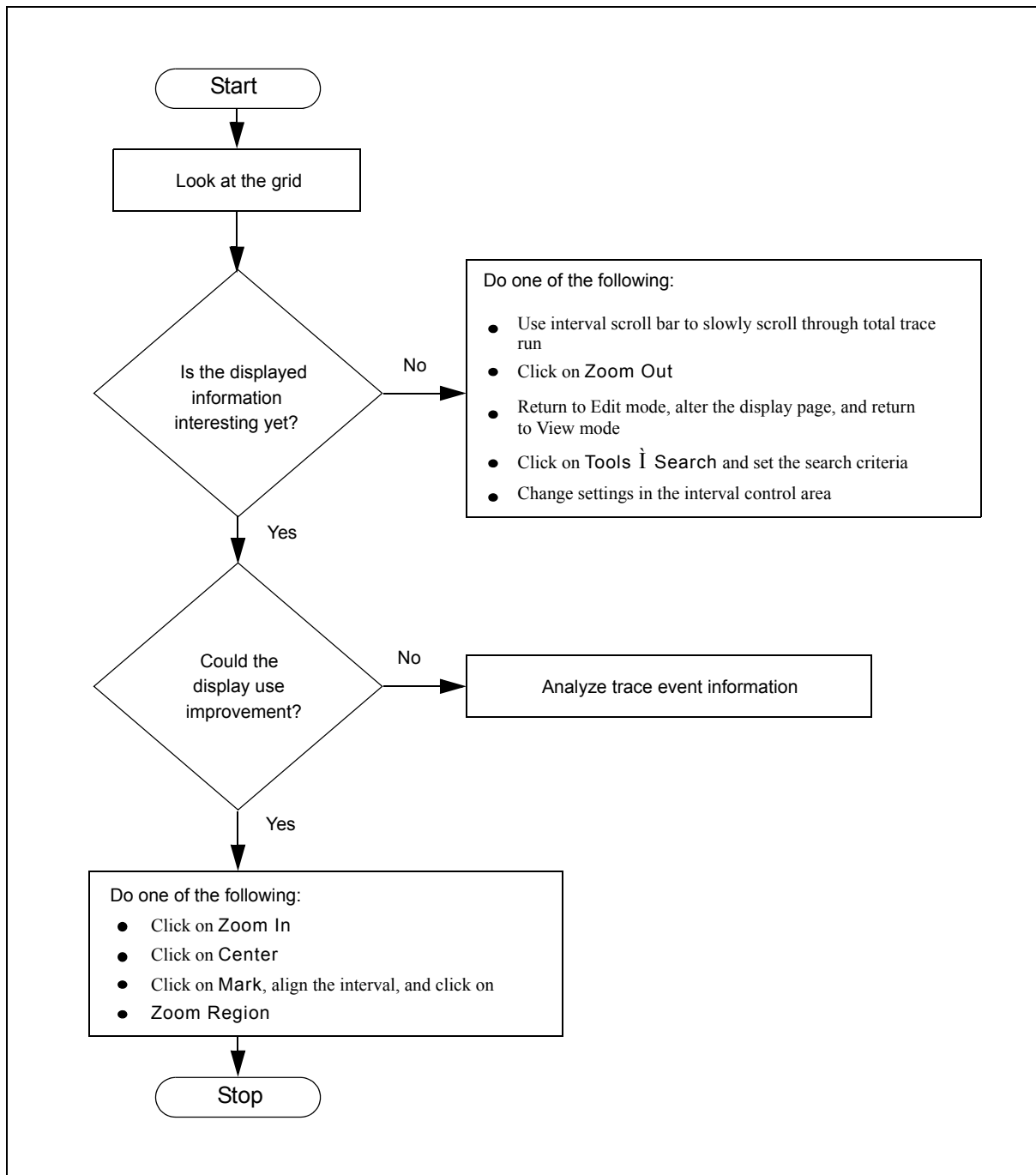


Figure 8-1. Deciding What to Do Next in View Mode

## Editing Single Fields

Changing the interval control area fields allows you to examine different intervals in your trace session. Usually you modify fields in the interval control area when you already know something about your trace events and their distribution.

When you press <Enter> or click on the **Apply** push button at the end of your editing, NightTrace validates the data in each field you modified and takes appropriate action. If NightTrace detects an invalid value, it restores the affected field to its previous value. For more information on the **Apply** push button, see “Interval Push Buttons” on page 9-31.

## Editing Multiple Fields

Sometimes it makes sense to change multiple fields for a single effect; for example, you may wish to change both the **Start Time** and **End Time** fields or you may wish to change both the **Start Time** and **Event Count** fields. In these cases, apply your changes only once, after you have edited each field of interest.

Changing some combinations of fields is not meaningful; for example, you may try to change both **Time Length** and **Event Count**. When NightTrace detects a meaningless combination of changes, it displays an error message in the message display area and restores the affected fields to their previous values. When NightTrace detects an invalid value, it restores the affected field to its previous value.

Some general rules apply to multiple field editing.

- You must not simultaneously apply changes to more than two trace event fields.
- You must not simultaneously apply changes to more than two time fields; for these purposes **Current Time** is not considered to be a time field.
- You can change **Current Time** with any other valid field changes as long as **Current Time** falls within the new interval.
- You can change **Zoom Factor** with any other valid field changes.
- You can change **Increment** with any other valid field changes.
- Simultaneously modifying one time field and clearing another time field makes NightTrace use the static and modified fields to determine the values of the cleared time field and the other fields.
- Simultaneously modifying one trace event field and clearing another trace event field makes NightTrace use the static and modified fields to determine the values of the cleared trace event field and the other fields.

The following table shows all the valid multiple field changes except those that involve **Current Time**, **Zoom Factor**, or **Increment**. For information on editing specific fields of the interval control area, see “Interval Control Area” on page 9-37.

**Table 8-2. Valid Multiple Field Changes**

Fields	Result
Start Time End Time	The new interval starts at <b>Start Time</b> and ends at <b>End Time</b> .
Start Time Time Length	The new interval starts at <b>Start Time</b> and has a length of the specified <b>Time Length</b> .
Time Length End Time	The new interval ends at <b>End Time</b> and has a length of the specified <b>Time Length</b> .
Start Event End Event	The new interval starts at ordinal trace event number (offset) <b>Start Event</b> and ends at ordinal trace event number (offset) <b>End Event</b> .
Start Event Event Count	The new interval starts at ordinal trace event number (offset) <b>Start Event</b> and includes the specified quantity of trace events.
Event Count End Event	The new interval ends at ordinal trace event number (offset) <b>End Event</b> and includes the specified quantity of trace events.
Start Time Event Count	The new interval starts at <b>Start Time</b> and includes the specified quantity of trace events unless the <b>Time Length</b> forces <b>Start Time</b> to change.
End Time Event Count	The new interval ends at <b>End Time</b> and includes the specified quantity of trace events unless the <b>Time Length</b> forces <b>End Time</b> to change.
Start Event Time Length	The new interval starts at ordinal trace event number (offset) <b>Start Event</b> and has a length of the specified <b>Time Length</b> unless the <b>Time Length</b> forces <b>Start Event</b> to change.
End Event Time Length	The new interval ends at ordinal trace event number (offset) <b>End Event</b> and has a length of the specified <b>Time Length</b> unless the <b>Time Length</b> forces <b>End Event</b> to change.



---

Default Display Page . . . . .	9-1
Menu Bar . . . . .	9-3
Page . . . . .	9-3
Edit . . . . .	9-5
Tags . . . . .	9-7
Edit String Tables . . . . .	9-9
Edit String Table . . . . .	9-13
Edit String Table Entry . . . . .	9-15
Edit Event Map Entry . . . . .	9-16
Create . . . . .	9-19
Actions . . . . .	9-21
Options . . . . .	9-24
Help . . . . .	9-26
Message Display Area . . . . .	9-28
Grid . . . . .	9-28
Interval Scroll Bar . . . . .	9-30
Interval Push Buttons . . . . .	9-31
Mode Button . . . . .	9-36
Interval Control Area . . . . .	9-37





A *display page* lets you view trace event data by allowing you to:

- create and configure display objects to graphically depict your trace session (see Chapter 10 “Display Objects”)
- examine trace events, trace event arguments, states, and timings using different display objects (see Chapter 8 “Viewing Trace Event Logs”)
- define macros, qualified events, and qualified states (see Chapter 11 “Using Expressions”) to aid in the analysis of trace data
- search for certain trace events based on specific criteria (see “Searching for Points of Interest” on page 12-1)
- summarize data into statistical information regarding particular trace events and states (see “Summarizing Statistical Information” on page 12-12)

## Default Display Page

The default display page contains a number of preconfigured display objects (see Chapter 10 “Display Objects”) that allow you to analyze your trace data with minimal effort. If this page does not exactly meet your needs, you can modify it according to your specifications. NightTrace brings up this page in *view mode* (see “Mode Button” on page 9-36 for more information).

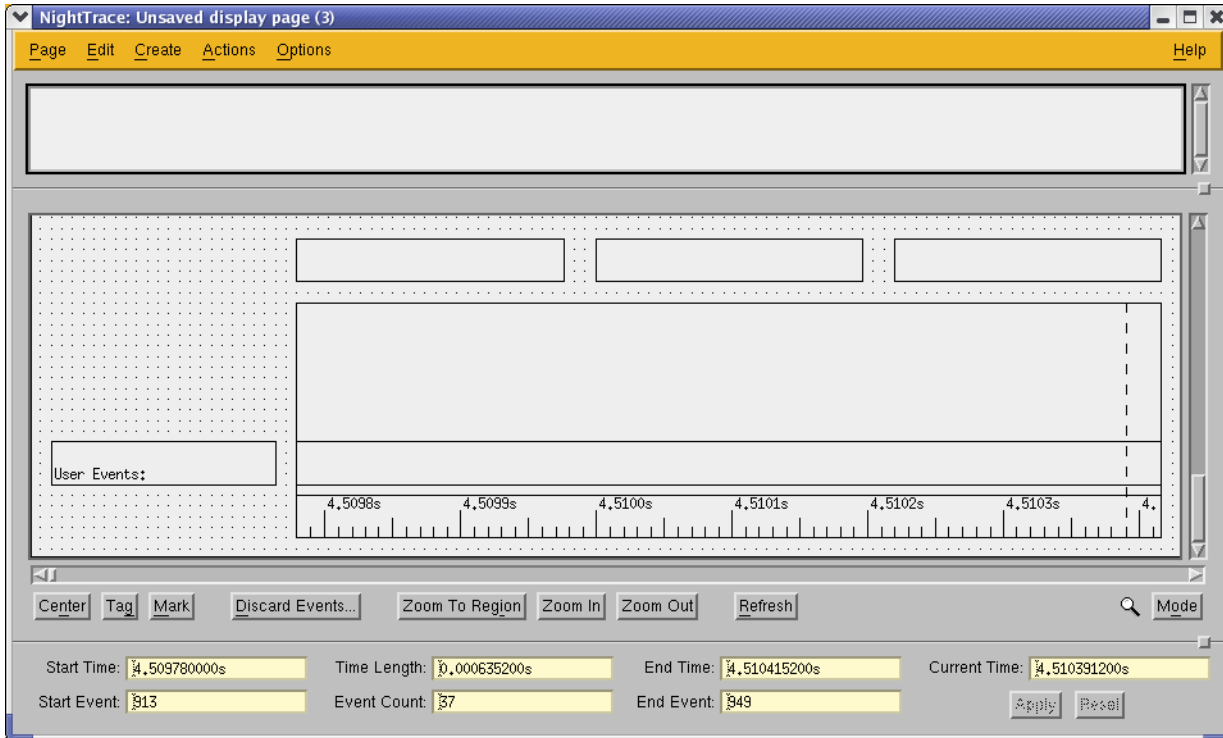
A default display page contains a Grid Label (see “Grid Label” on page 10-4) and a State Graph (see “State Graph” on page 10-7) for each thread logging trace events in your trace event file(s). Each State Graph is configured to display only those events logged by a particular thread; the associated Grid Label identifies that thread. An additional State Graph is also created which is configured to display all user events from all threads combined. If the number of threads is so large that their associated State Graphs will not all fit on the grid, then NightTrace does not display any State Graphs.

In addition, Data Boxes (see “Data Box” on page 10-5) appear at the top of the default display page containing information related to the *current trace event*. This information includes the *offset*, *trace event ID*, and first *trace event argument* logged by that particular trace event.

When analyzing trace event files from multiple systems, if a thread name is not unique in the trace events, NightTrace prints a node name along with the process ID number and thread ID number in the associated Grid Label to identify that thread.

Figure 9-1 shows a default display page for two threads, `jane` and `tarzan`, logging trace events. The information in the Data Boxes at the top of the grid relate to the last trace

event on or before the current time line. A State Graph has been created showing the trace events logged by the thread jane; another has been created showing those logged by tarzan. A third State Graph appears below the others displaying the trace events logged by both threads.



**Figure 9-1. A Default Display Page**

A display page consists of the following components:

- Menu Bar (see “Menu Bar” on page 9-3)
- Message Display Area (see “Message Display Area” on page 9-28)
- Grid (see “Grid” on page 9-28)
- Interval Scroll Bar (see “Interval Scroll Bar” on page 9-30)
- Interval Push Buttons (see “Interval Push Buttons” on page 9-31)
- Interval Control Area (see “Interval Control Area” on page 9-37)

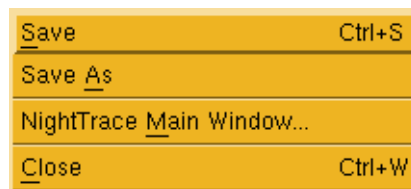
## Menu Bar

The menu bar on all display pages provides access to the following menus:

- **Page** (see “Page” on page 9-3)
- **Edit** (see “Edit” on page 9-5)
- **Create** (see “Create” on page 9-19)
- **Actions** (see “Actions” on page 9-21)
- **Options** (see “Options” on page 9-24)
- **Help** (see “Help” on page 9-26)

## Page

The **Page** menu appears on the menu bar of all display pages (see “Menu Bar” on page 9-3).



**Figure 9-2. Display Page - Page menu**

### Save

Saves the current display page configuration (see “Page Configuration Files” on page 4-13) to the external file specified with the **Save As...** menu item. Any changes you have made since the last save operation will be saved to that file; this menu item is disabled (desensitized) if no changes have been made.

This menu item is also disabled if this is a new display page; in this case, use the **Save As ....** menu item to specify a filename.

### Save As...

Presents a file selection dialog to specify a filename to which the current display page configuration will be saved (see “Page Configuration Files” on page 4-13).

**NightTrace Main Window...**

Opens the NightTrace Main Window if not currently opened; otherwise, brings the NightTrace Main Window to the foreground.

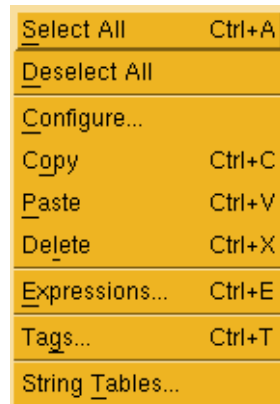
See Chapter 5 “Using the NightTrace GUI” for more information.

**Close**

Ends the current editing/viewing session, resets all field and radio button settings, and clears the message display area. If you have unsaved changes, a warning dialog box appears, asking if you want to save your changes.

## Edit

The Edit menu appears on the menu bar of all display pages (see “Menu Bar” on page 9-3).



**Figure 9-3. Display Page - Edit menu**

### Select All

Selects every display object on the grid. This is useful when you want to perform some operation on every display object on the grid (for example, moving or deleting every display object).

#### NOTE

This operation is enabled only when the display page is in *edit mode* (see “Mode Button” on page 9-36).

### Deselect All

Deselects every selected display object on the grid.

#### NOTE

This operation is enabled only when the display page is in *edit mode* (see “Mode Button” on page 9-36).

### Configure...

Opens the configuration dialog(s) for the selected display object(s).

**NOTE**

Double-clicking on a particular display object will bring up the configuration dialog for that display object.

See “Configuring Display Objects” on page 10-15 for details.

This operation is enabled only when the display page is in *edit mode* (see “Mode Button” on page 9-36).

**Copy**

Accelerator: **Ctrl+C**

Copy the selected display object allowing the user to subsequently paste a copy of the display object on the Grid using the **Paste** menu item.

Only one display object may be copied at a time. In addition, Column display objects (see “Column” on page 10-6) cannot be copied.

**NOTE**

This operation is enabled only when the display page is in *edit mode* (see “Mode Button” on page 9-36).

**Paste**

Accelerator: **Ctrl+V**

When this menu item is selected, the mouse pointer becomes a crosshair allowing the user to position and size the display object previously copied using the **Copy** menu item.

See “Creating Display Objects” on page 10-12 for more information.

**NOTE**

This operation is enabled only when the display page is in *edit mode* (see “Mode Button” on page 9-36).

**Delete**

Deletes the selected display object(s).

This operation is enabled only when the display page is in *edit mode* (see “Mode Button” on page 9-36).

### Expressions...

Opens the NightTrace Qualified Expressions dialog (see “NightTrace Qualified Expressions” on page 11-119) allowing the user to create or edit qualified expressions such as qualified states, qualified events, and macros.

### Tags...

Opens the Tags dialog (see “Tags” on page 9-7) which provides a summary of all tags in the current session. This dialog also allows for the manipulation of tags, determining time differences between two selected tags, and seeing the time difference between each tag and the *current time line* on the display page.

### String Tables...

Opens the Edit String Tables dialog (see “Edit String Tables” on page 9-9) which provides a list of current string tables by name as well as the number of entries in each table. This dialog also allows you to add new string tables, and edit or remove existing string tables.

## Tags

The Tags dialog is opened by selecting the Tags... menu item (see “Tags...” on page 9-7) from the Edit menu of any display page.

The Tags dialog provides a summary of all tags in the current session. This dialog also allows for the manipulation of tags, determining time differences between two selected tags, and seeing the time difference between each tag and the *current time line* on the display page.

See “Tag” on page 9-32 for more information.

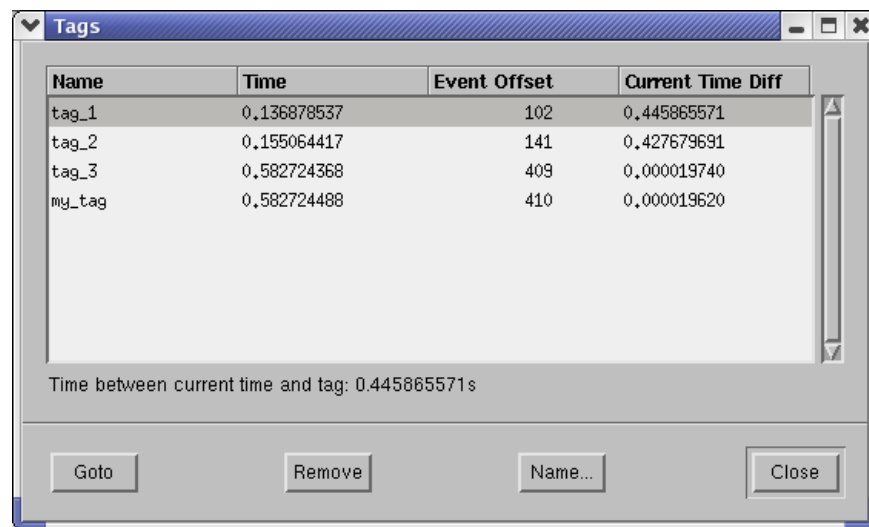


Figure 9-4. Tags dialog

To see the time difference between two tags, select the first tag of interest and then, holding the Ctrl key, select the second tag of interest. If only one tag is selected, the time difference between the selected tag and the current time is displayed.

**Name**

The name of the tag as entered in the **Create New Tag** dialog or a default name as generated by NightTrace.

**Time**

The time on the Ruler where the tag can be found; for tagged events, this corresponds to the event time.

**Event Offset**

For tagged events, this is the *offset* of the event tagged; for tagged times, this is the offset of the last event before the tag.

**Current Time Diff**

Shows the difference in seconds between the *current time line* on the display page and the time associated with the tag.

The following buttons appear at the bottom of the **Tags** dialog:

**Goto**

Applies to one selected tag; places the *current time line* on the selected tag on the display page; this can also be accomplished by double-clicking on the tag in the list.

**Remove**

Applies to one or more selected tags; permanently removes the tag(s) from the display page.

**Name**

Opens the **Set Tag Name** dialog as shown in Figure 9-5 allowing the user to change the name of the tag selected in the **Tags** dialog.





**Figure 9-5. Set Tag Name dialog**

### **Close**

Dismisses the Tags dialog.

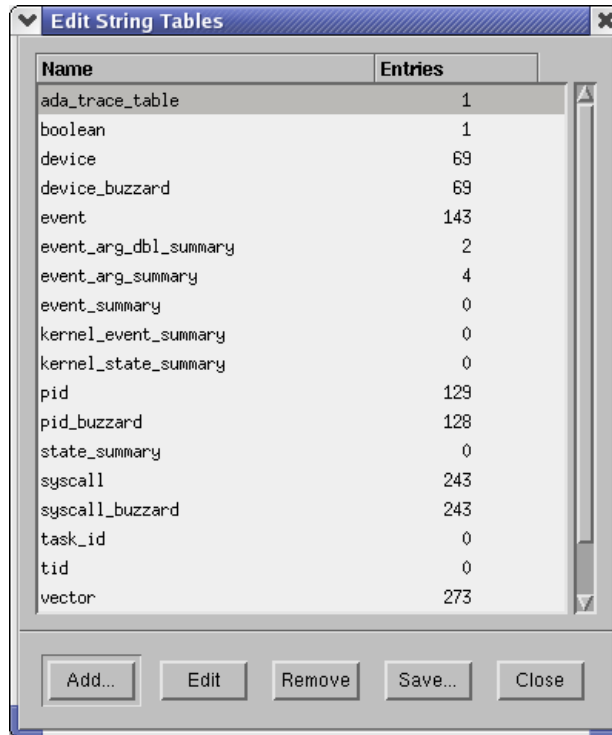
### **Help**

Displays online help for the Tags dialog.

## **Edit String Tables**

The **Edit String Tables** dialog is opened by selecting the **String Tables...** menu item (see “String Tables...” on page 9-7) from the **Edit** menu of any display page. In addition, this dialog can be accessed by double-clicking the string tables entry in the Session Overview Area or by pressing the **Edit...** button while the string tables entry is selected in the Session Overview Area (see “Session Overview Area” on page 5-39).

The **Edit String Tables** dialog provides a list of current string tables alphabetized by name as well as the number of entries in each table.



**Figure 9-6. Edit String Tables dialog**

**Add...**

Presents the Add Table dialog as shown in Figure 9-7 allowing the user to specify the name of the new string table.



**Figure 9-7. Add Table dialog**

After it has been added, the new string table will appear in the alphabetized list in the Edit String Tables dialog.

**Edit**

Opens the **Edit String Table** dialog (see “Edit String Table” on page 9-13) which allows you to edit the selected string table.

**NOTE**

You may double-click on any item in the list to edit that individual string table.

**Remove**

Removes the selected table(s).

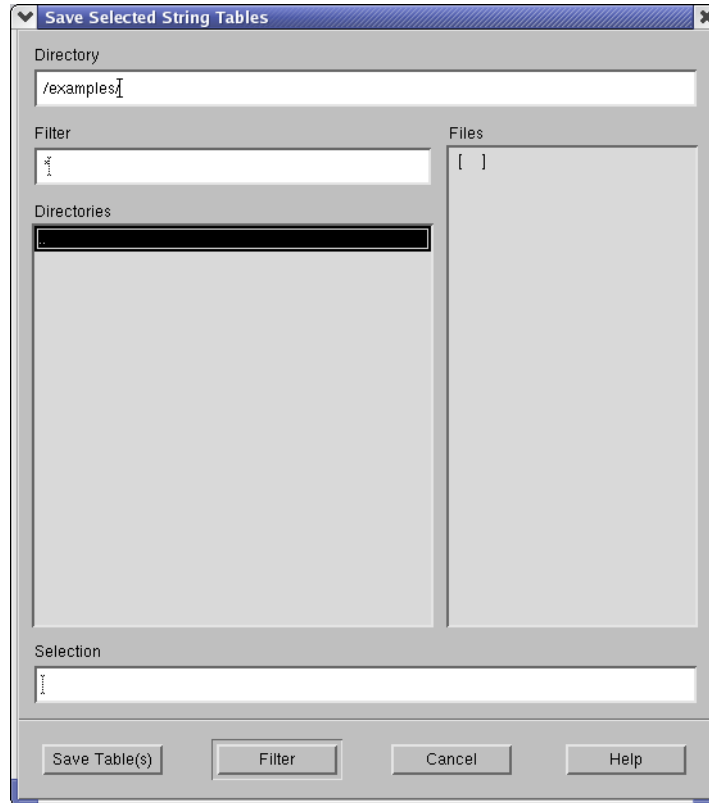
**NOTE**

Outstanding references to removed tables remain in the session, possibly resulting in error messages in display pages, searches, and summaries.

In addition, tables populated by default by NightTrace (e.g. `event`, `boolean`) may not be removed; NightTrace silently ignores the remove request for such tables.

**Save**

Presents the **Save Selected String Tables** file selection dialog as shown in Figure 9-8 allowing the user to save the selected table(s) to an ASCII NightTrace configuration file for use in other NightTrace sessions.



**Figure 9-8. Save Selected String Tables dialog**

To save the table(s), specify a filename in the Save Selected String Tables file selection dialog and press the Save Table(s) button.

**NOTE**

Only the selected string tables are saved.

The tables in this saved file may be imported into NightTrace by:

- specifying the file as an argument on the command line when starting NightTrace (see Chapter 4 “Invoking NightTrace”)
- opening the file in a NightTrace session by pressing the Open... button below the Session Overview Area (see “Session Overview Area” on page 5-39)

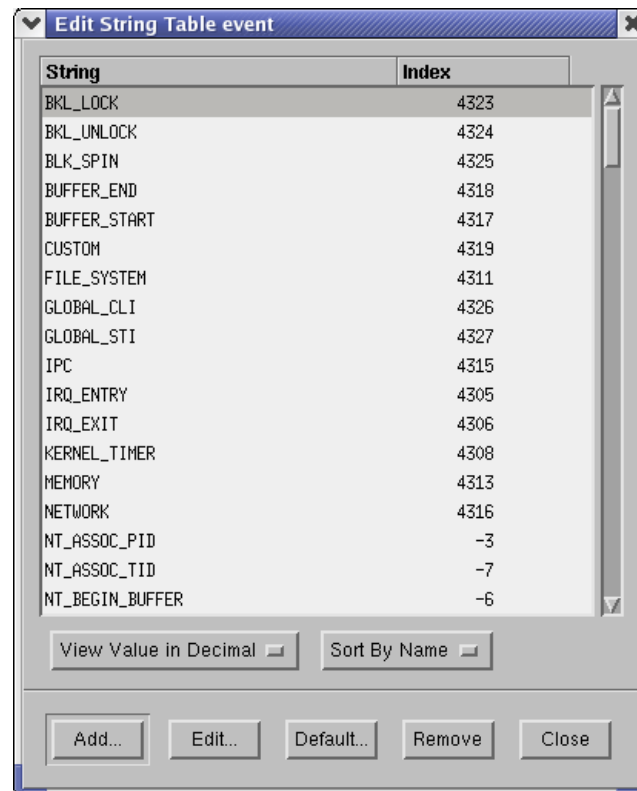
**Close**

Dismisses the Edit String Tables dialog.

## Edit String Table

The **Edit String Table** dialog lists each string representation and its associated integer value in a particular string table and allows you to add, edit, or remove entries from that string table.

The **Edit String Table** dialog is opened by double-clicking the desired string table entry in the **Edit String Tables** dialog (see “**Edit String Tables**” on page 9-9) or by pressing the **Edit...** button while the desired string table entry is selected in the **Edit String Tables** dialog.



**Figure 9-9. Edit String Table dialog**

### View Value

Provides two options for viewing integer values in table:

#### View Value in Decimal

Displays each integer value in decimal representation.

#### View Value in Hexadecimal

Displays each integer value in hexadecimal representation.

### **Sort By**

Provides two options for sorting the table entries:

#### **Sort by Name**

Lists the table entries in alphabetical order according to name.

#### **Sort by Value**

Lists the table entries in numerical order according to value.

### **Add...**

Presents the **Edit String Table Entry** dialog (see “Edit String Table Entry” on page 9-15) allowing the user to add an entry to the current string table.

#### **NOTE**

When adding to the `event` string table, the **Edit Event Map Entry** dialog is presented (see “Edit Event Map Entry” on page 9-16).

### **Edit...**

Presents the **Edit String Table Entry** dialog (see “Edit String Table Entry” on page 9-15) allowing the user to edit the selected entry in the current string table.

#### **NOTE**

When editing an entry in the `event` string table, the **Edit Event Map Entry** dialog is presented (see “Edit Event Map Entry” on page 9-16).

### **Default...**

Presents the **Edit String Table Entry** dialog allowing the user to edit the default string to use when the `get_string()` function (see “`get_string()`” on page 11-104) is passed an integer value that is not mapped to a string in the table.

### **Remove**

Permanently removes selected table entries.

### **Close**

Dismisses the **Edit String Table** dialog.

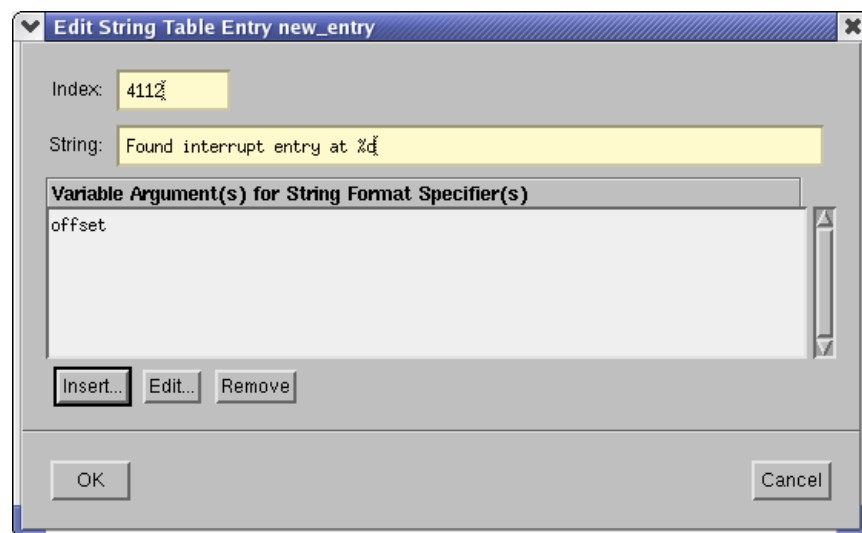
## Edit String Table Entry

The **Edit String Table Entry** dialog allows the user to add a new entry or edit an existing entry in a particular string table.

See “Edit String Table” on page 9-13 to add, edit, or remove other entries in a string table.

### NOTE

When adding or editing an entry in the `entry` string table, Night-Trace presents the **Edit Event Map Entry** dialog (see “Edit Event Map Entry” on page 9-16).



**Figure 9-10. Edit String Table Entry dialog**

### Index

Numerical value of the string table entry.

### String

String representation for the entry value.

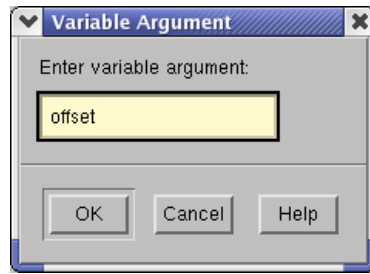
The specified string may be a formatted string with format specifiers (see “`get_format()`” on page 11-108). Arguments associated with these format specifiers are added to this list using the **Insert** button on this dialog.

Figure 9-10 shows a **String** with a `%d` format specifier. The NightTrace expression (see “Expressions” on page 11-1) associated with that specifier, `offset`, can be seen in the **Variable Argument(s) for String Format Specifier(s)** list.

### Insert

Presents the **Variable Argument** dialog as shown in Figure 9-11 allowing the user to associate a NightTrace expression (see “Expressions” on page 11-1) with a format specifier used in the **String**.

Figure 9-10 shows a **String** with a `%d` format specifier. The argument associated with that specifier, `offset`, can be seen in the **Variable Argument(s) for String Format Specifier(s)** list.



**Figure 9-11. Variable Argument dialog**

### Edit

Presents the **Variable Argument** dialog as shown in Figure 9-11 allowing the user to edit the NightTrace expression (see “Expressions” on page 11-1) selected in the **Variable Argument(s) for String Format Specifier(s)** list.

### Remove

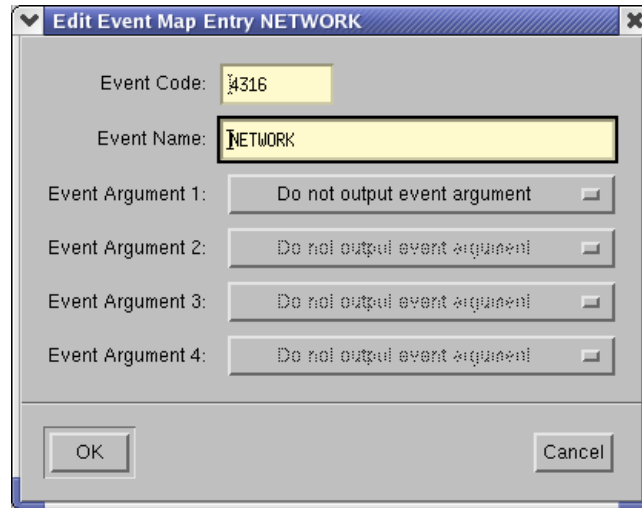
Removes the selected argument from the **Variable Argument(s) for String Format Specifier(s)** list.

## Edit Event Map Entry

The **Edit Event Map Entry** dialog allows the user to add a new entry or edit an existing entry in the `event` string table.

See “Edit String Table” on page 9-13 to add, edit, or remove other entries in a string table.





**Figure 9-12. Edit Event Map Entry dialog**

### Event Code

A valid integer in the range reserved for user trace events (0-4095, inclusive).

### Event Name

A character string to be associated with the user trace event specified in Event Code.

Trace event names must begin with a letter and consist solely of alphanumeric characters and underscores.

In addition, the user may specify up to four arguments to display for the user defined trace event. These arguments are displayed when:

- **ntrace** is invoked with the **--listing (-l)** option (see “-l --listing” on page 4-2)
- the user middle clicks on an event in the display page grid (shown in the Message Display Area)
- when an event is found via the search mechanism (see “Searching for Points of Interest” on page 12-1), the event description is given in the Message Display Area (see “Message Display Area” on page 9-28)

The following items allow the user to choose the base format in which to display each argument. The format is specified by choosing one of the following from the drop-down associated with that argument:

- Output event argument as float
- Output event argument as decimal
- Output event argument as hexadecimal
- Output event argument as float

#### **Event Argument 1**

Specifies whether the first argument is to be displayed and the base format to display that argument.

#### **Event Argument 2**

Specifies whether the second argument is to be displayed and the base format to display that argument.

#### **Event Argument 3**

Specifies whether the third argument is to be displayed and the base format to display that argument.

#### **Event Argument 4**

Specifies whether the fourth argument is to be displayed and the base format to display that argument.

Each combo box only allows for reasonable input as per the NightTrace API calls (see "Understanding NightTrace Library Calls" on page 2-3). For instance:

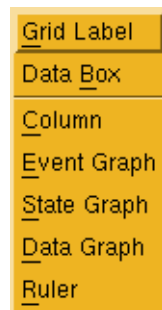
- if an event argument is displayed, all prior arguments must be displayed (i.e. cannot display Event Argument 2 if you are not displaying Event Argument 1)
- if Event Argument 1 is output as float, Event Argument 2 must be output as float or not output
- if Event Argument 2 is output as float, Event Argument 1 must be output as float

## Create

The **Create** menu appears on the menu bar of all display pages (see “Menu Bar” on page 9-3) and allows you to add display objects to a display page.

### NOTE

The display page must be in *edit mode* in order to use these selections (see “Mode Button” on page 9-36 for more information).



**Figure 9-13. Display Page - Create menu**

### Grid Label

Allows the user to add a Grid Label to the current display page.

See “Grid Label” on page 10-4 for more information.

### Data Box

Allows the user to add a Data Box to the current display page.

See “Data Box” on page 10-5 for more information.

### Column

Allows the user to add a Column to the current display page.

See “Column” on page 10-6 for more information.

### Event Graph

Allows the user to add a Event Graph to the current display page.

See “Event Graph” on page 10-6 for more information.

**State Graph**

Allows the user to add a State Graph to the current display page.

See “State Graph” on page 10-7 for more information.

**Data Graph**

Allows the user to add a Data Graph to the current display page.

See “Data Graph” on page 10-8 for more information.

**Ruler**

Allows the user to add a Ruler to the current display page.

See “Ruler” on page 10-10 for more information.

## Actions

The **Actions** menu appears on the menu bar of all display pages (see “Menu Bar” on page 9-3).

<u>C</u> hange Search Criteria...	Ctrl+F
Search <u>F</u> orward	Period
Search <u>B</u> ackward	Comma
Change <u>S</u> ummary Criteria...	Ctrl+Z
<u>S</u> ummarize	Ctrl+U
Zoom <u>I</u> n	Down-Arrow
Zoom <u>O</u> t	Up-Arrow
Scroll <u>F</u> orward	Right-Arrow
Scroll <u>B</u> ackward	Left-Arrow

**Figure 9-14. Display Page - Actions menu**

### Change Search Criteria...

Accelerator: Ctrl+F

Opens the **Search NightTrace Events** dialog, allowing the user to locate areas of interest in their trace event file(s)

See “Searching for Points of Interest” on page 12-1 for more information.

### Search Backward

Accelerator: <

#### NOTE

It is not necessary to press the **Shift** key when using this accelerator.

Furthermore, it is not necessary to have the **Search NightTrace Events** window open when using this accelerator (see “Searching for Points of Interest” on page 12-1). The search criteria specified from the previous search is used.

Attempts to find the first trace event occurring *before* the current time line that matches the search criteria.

See “Searching for Points of Interest” on page 12-1 for more information.

## Search Forward

Accelerator: >

### NOTE

It is not necessary to press the **Shift** key when using this accelerator.

Furthermore, it is not necessary to have the **Search NightTrace Events** window open when using this accelerator (see “Searching for Points of Interest” on page 12-1). The search criteria specified from the previous search is used.

Attempts to find the next trace event occurring *after* the current time line that matches the search criteria.

See “Searching for Points of Interest” on page 12-1 for more information.

## Change Summary Criteria...

Accelerator: Ctrl+Z

Opens the **Summarize NightTrace Events** dialog, allowing the user to locate areas of interest in their trace event file(s)

See “Summarizing Statistical Information” on page 12-12 for more information.

## Summarize

Accelerator: Ctrl+U

Performs a summary of the information in the current trace event file based on the criteria specified in the **Summarize NightTrace Events** dialog.

See “Summarizing Statistical Information” on page 12-12 for more information.

## Zoom In

Accelerator: down-arrow

Reduces the interval by the **Zoom Factor** (see “Zoom Factor...” on page 9-24) providing a more detailed view of the smaller interval; the interval is centered around the current time line.

Functionality is identical to that of the **Zoom In** button at the bottom of the display page (see “Zoom In” on page 9-35 for more details).

## **Zoom Out**

Accelerator: **up-arrow**

Enlarges the interval by the **Zoom Factor** (see “Zoom Factor...” on page 9-24) providing a higher-level view of the larger interval; the interval is centered around the current time line.

Functionality is identical to that of the **Zoom Out** button at the bottom of the display page (see “Zoom Out” on page 9-35 for more details).

## **Scroll Forward**

Accelerator: **right-arrow**

Scrolls the interval forward **Increment** seconds or **Increment** percent of the current display interval allowing you to examine different intervals in your trace session (see “Increment...” on page 9-24).

See “Interval Scroll Bar” on page 9-30 for related information.

## **Scroll Backward**

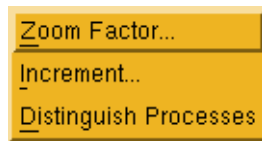
Accelerator: **left-arrow**

Scrolls the interval backward **Increment** seconds or **Increment** percent of the current display interval allowing you to examine different intervals in your trace session (see “Increment...” on page 9-24).

See “Interval Scroll Bar” on page 9-30 for related information.

## Options

The **Options** menu appears on the menu bar of all display pages (see “Menu Bar” on page 9-3).



**Figure 9-15. Display Page - Options menu**

### **Zoom Factor...**

The number of times to magnify (or reduce) the interval each time you click on **Zoom Out** (or **Zoom In**). The default is 2. (See “Zoom Out” on page 9-35 and “Zoom In” on page 9-35 for more information about these buttons.)

A valid change keeps **Zoom Factor** greater than or equal to 1. If you set **Zoom Factor** to the word `default` or a space, NightTrace resets **Zoom Factor** to the default value.

### **Increment...**

Controls how much the current interval scrolls (and the slider moves) when you:

- click on an arrowhead of the interval scroll bar (see “Interval Scroll Bar” on page 9-30)
- click between an arrowhead and the slider on the interval scroll bar
- select either the **Scroll Forward** or **Scroll Backward** menu item from the **Actions** menu of any display page (see “Actions” on page 9-21)
- use the < or > accelerator keys to scroll forward or backward (Note that it is not necessary to press the **Shift** key when using these accelerators.)

This field may contain either a percentage or an absolute amount of time in seconds. The default is 25%.

A valid change keeps percentages greater than 0% and less than or equal to 100% and absolute numbers greater than 0 microseconds and less than or equal to the end time of the trace session. If you set **Increment** to the word `default` or a space, NightTrace resets **Increment** to the default value.

If **Increment** is less than 100% when you click on an interval scroll bar arrowhead, you see part of the previous interval in this interval; if **Increment** is equal to 100%, you see a completely new interval.





**Figure 9-16. Amount of Scrolling Due to Increment Value**

### Distinguish Processes

Distinguishes process names by appending PID specific information to the process name in the string tables `pid`, `pid_` (see “Pre-Defined String s” on page 4-16).

For example, if the trace data contains events from two different processes named `top`, and the first process has a PID of 1633 and the second has a PID of 18957, the following would appear when listing the process names in the various NightTrace dialogs:

```
top_1633
top_18957
```

For instance, these process names would appear in the **Value** drop-down in the **Search NightTrace Events** dialog (see “Searching for Points of Interest” on page 12-1) when the **Key** is set to `Process ID`, allowing the user to differentiate between the two processes.

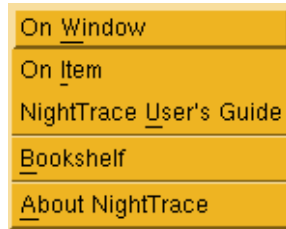
On PowerMAX OS systems, the suffix:

```
_1wpn           where n is a unique number
```

is appended to the end of the process name to differentiate between multiple light-weight processes.

## Help

The **Help** menu appears on the menu bar of all display pages (see “Menu Bar” on page 9-3).



**Figure 9-17. Display Page - Help menu**

### On Window

Opens the help topic for the current window.

### On Item

Gives context-sensitive help on the various menu options, dialogs, or other parts of the user interface.

Help for a particular item is obtained by first choosing this menu option, then clicking the mouse pointer on the object for which help is desired (the mouse pointer will become a floating question mark when the **On Item** menu item is selected).

In addition, context-sensitive help may be obtained for the currently highlighted option by pressing the **F1** key. The HyperHelp viewer will open with the appropriate topic displayed.

### NightTrace User's Guide

Opens the online version of the *NightTrace User's Guide* (0890398) in the HyperHelp viewer.

The online *NightTrace User's Guide* can also be accessed using the **nhelp** utility shipped with the X Window System. The manual name is **ntrace**. For example, from the command line:

```
nhelp ntrace
```

opens the most recently installed version of the *NightTrace User's Guide* in the HyperHelp viewer.

**Bookshelf**

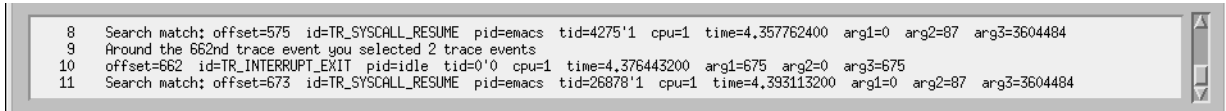
Opens a HyperHelp window that lists all of the currently available HyperHelp publications.

**About NightTrace**

Displays version and copyright information for the NightTrace product.

## Message Display Area

The Message Display Area presents various diagnostic and informational messages. Figure 9-18 shows some of these types of messages in a Message Display Area.



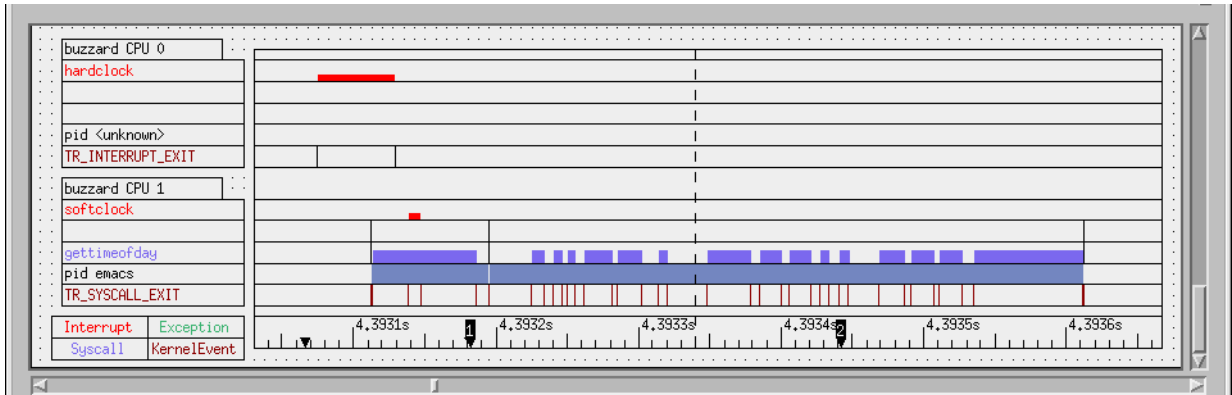
**Figure 9-18. Message Display Area**

The Message Display Area can include such messages as:

- error messages (e.g. from incorrect values entered in configuration dialogs)
- detailed textual information about specific events (see “Grid” on page 9-28)
- the time between the current time line and the mouse cursor (by pressing mouse button 3 at a particular point on the grid)
- the time between the mouse cursor and the mark (see “Mark” on page 9-33)
- results of search operations (see “Searching for Points of Interest” on page 12-1)
- results of summary operations (see “Summarizing Statistical Information” on page 12-12)

## Grid

The *grid* is a region of the display page that is filled with parallel rows and columns of dots. These dots serve as reference points for display-object alignment. You can alter the grid dimensions by changing the size of the display page. To change the display page size, resize your window by using features of your window manager.



**Figure 9-19. The Grid**

NightTrace assigns each trace event in the trace session a unique ordinal number or *offset* beginning with ordinal number 0. These ordinal numbers appear in the interval control area and in the message display area. For more information on ordinal trace events, see “Interval Control Area” on page 9-37.

Some display objects on the grid contain vertical lines. Each vertical line in a State Graph (see “State Graph” on page 10-7) or Event Graph (see “Event Graph” on page 10-6) represents one or more user trace events, kernel trace events, or NightTrace internal trace events. If more than one event is represented by a vertical line, zooming in will provide sufficient resolution to display each trace event as a separate vertical line (see “Zoom In” on page 9-35).

If you click on a trace event with mouse button 2, NightTrace writes information about that trace event in the message display area. Each vertical line in a Data Graph (see “Data Graph” on page 10-8) represents a trace event argument. If you click on a data value with mouse button 3, NightTrace writes information about the data value in the message display area.

If your grid has a Column (see “Column” on page 10-6) and you have not already positioned your interval somewhere else, NightTrace displays in the Column the earliest 5 percent of your trace session. Usually this information is uninteresting and you want to see other parts of your trace session. The following list shows the ways you can get NightTrace to locate interesting parts of your trace session:

- Scroll through the interval using the interval scroll bar
- Zoom in or zoom out using interval push buttons
- Change the parameters defining the interval by editing its fields
- Use the **Search NightTrace Events** dialog to search for a specific trace event or condition. (See “Searching for Points of Interest” on page 12-1 for more information.)

## Interval Scroll Bar

Moving the slider of the interval scroll bar allows you to examine different intervals in your trace session. By moving the slider, you change the displays in display objects on the grid and in the interval control area (see “Interval Control Area” on page 9-37). Changes in the display objects are most obvious when you have a Column that contains both a State Graph and a Ruler. See Chapter 10 “Display Objects” for more information on display objects.

The interval scroll bar is horizontal and extends the entire width of the grid. The left arrowhead represents the beginning of the entire trace session, not just the part displayed on the grid or by the interval control area fields. The right arrowhead represents the end of the entire trace session.

If you have not already positioned your interval somewhere else, the movable slider of the interval scroll bar is adjacent to the scroll bar's left arrowhead. When the slider is here, the **Start Time** statistic in the interval control area is 0.0000000 seconds. The length of the slider is proportionate to the amount of the trace session displayed in the interval. By default, a display page shows 5% of a trace session.

In the following interval scroll bar descriptions, the fields in the interval control area that are affected by the interval scroll bar include: **Current Time**, **Start Time**, **End Time**, **Start Event**, **End Event**, and **Increment**. For more information on these fields, see “Interval Control Area” on page 9-37.

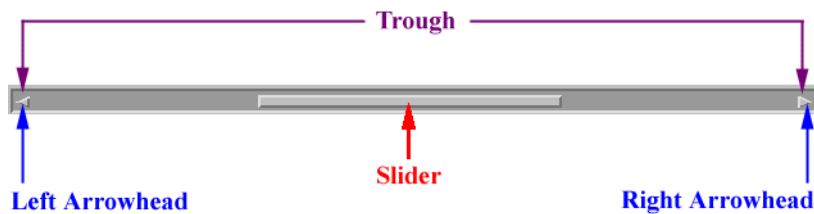


Figure 9-20. The Interval Scroll Bar

Manipulating the interval scroll bar in the following ways has the following results.

**Table 9-1. Manipulating the Interval Scroll Bar**

Action	Mouse Button	Location	Result
Click	Any	Left arrowhead	If the interval scroll bar slider is not already at the leftmost position: <ul style="list-style-type: none"> <li>• Moves the slider to the left.</li> <li>• Scrolls backward <b>Increment</b> seconds or <b>Increment</b> percent of the current display interval.</li> </ul>
Click	Any	Right arrowhead	If the interval scroll bar slider is not already at the rightmost position: <ul style="list-style-type: none"> <li>• Moves the slider to the right.</li> <li>• Scrolls forward <b>Increment</b> seconds or <b>Increment</b> percent of the current display interval.</li> </ul>
Click	1	Between an arrowhead and the slider	<ul style="list-style-type: none"> <li>• Moves the slider to the side you clicked on.</li> <li>• Scrolls the current interval by twice the number of seconds in <b>Increment</b> or by twice the percentage in <b>Increment</b>.</li> </ul>
Click or Drag	2	Between an arrowhead and the slider	<ul style="list-style-type: none"> <li>• Moves the slider where you clicked and/or dragged.</li> <li>• Scrolls the current interval accordingly.</li> <li>• If your current time line was not centered, centers it.</li> </ul>
Drag	1 or 2	Slider	(Same as preceding entry.)
Press and Hold	Any	Left or right arrowhead	Causes animated scrolling of data in the direction the arrow points

## Interval Push Buttons

The interval push buttons let you examine different intervals in your trace session. The eight push buttons appear just below the Grid (see “Grid” on page 9-28) on the display page.



**Figure 9-21. Interval Push Buttons**

### Center

Centers the interval around the current time line in a Column (see "Column" on page 10-6).

Makes corresponding changes to Start Time, End Time, Start Event, and End Event.

### Tag

Places a *tag* (represented by a uniquely-numbered marker) at the *current time line* on the Ruler (see "Ruler" on page 10-10). The tag is associated with the last event before the tag. The name of the tag is automatically generated by NightTrace with the form:

tag\_ *n*

where *n* is a unique number.

Tags can be useful for:

- remembering or identifying points of interest in analyzing trace data
- allowing quick traversal of points of interest via the search mechanism (see "Searching for Points of Interest" on page 12-1)
- determining time differences between points of interest
- simplifying analysis of the same data across multiple invocations of NightTrace due to the fact that named tags are saved with the session (see "Session Configuration Files" on page 4-24)

To tag a particular event, select the event with the third mouse button. The **Create New Tag** dialog is presented as shown in Figure 9-22:



**Figure 9-22. Create New Tag dialog**

Tag names appear in the **Value** drop-down menus in the various NightTrace dialogs when the **Key** is set to **Tagged Event**. This can be useful when searching for a particular tagged event (see "Searching for Points of Interest" on page 12-1).

The tag name can be changed by using the **Tags** dialog (see "Tags" on page 9-7).



**NOTE**

Named tags are associated with the trace data that was tagged. Discarding trace events through the **Discard Events...** button or closing a trace data segment via the **Close** button in the Session Overview removes the tags (see “Session Overview Area” on page 5-39).

**Mark**

Places a *mark* (represented by a solid triangle) at a particular time on the Ruler (see “Ruler” on page 10-10). The mark defaults to time 0.

The mark references a particular position and may be used to determine distances from the *current time line* as well as from any position where the mouse pointer is clicked. The area between the mark and the current time line is referred to as a *region* and is used by both the **Zoom To Region** (see “Zoom To Region” on page 9-35) and **Discard Events** (“Discard Events...” on page 9-34) features of NightTrace.

**NOTE**

NightTrace currently supports only one mark.

Simultaneously pressing **Ctrl** and clicking on mouse button 1 moves the mark and the current time line to the place where the mouse is pointing.

Simultaneously holding down **Ctrl** and clicking on mouse button 2 causes NightTrace to write a statistic in the message display area (see “Message Display Area” on page 9-28) that tells the distance (in seconds) that the mouse pointer is from the mark. If the mouse pointer is to the right of the mark, the number will have a positive value; if the mouse pointer is to the left of the mark, the number will have a negative value.

Simultaneously holding down **Ctrl** and clicking on mouse button 3 causes NightTrace to write a statistic in the message display area that tells the distance (in seconds) that the mouse pointer is from the current time line. If the mouse pointer is to the right of the current time line, the number will have a positive value; if the mouse pointer is to the left of the current time line, the number will have a negative value.

Simultaneously holding down **Ctrl**, holding down mouse button 1, and dragging the mouse pointer horizontally in a Column (see “Column” on page 10-6) moves the mark to the beginning point of the drag region and moves the current time line to the ending point of the drag region. The region is highlighted as you drag the pointer. This action is useful when using the **Zoom To Region** (see “Zoom To Region” on page 9-35) or **Discard Events** (“Discard Events...” on page 9-34) features.

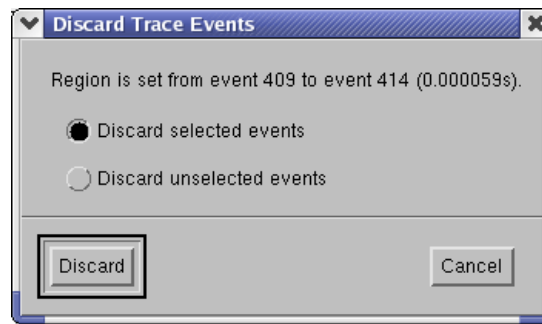
### Discard Events...

Presents the Discard Trace Events dialog allowing the user to reduce the set of trace data to either the region between the *mark* and the *current time line* (inclusive) or the trace data excluding that region.

#### NOTE

See “Mark” on page 9-33 for more information on specifying a region.

The Discard Trace Events dialog is shown in Figure 9-23:



**Figure 9-23. Discard Trace Events dialog**

The trimmed data set may be saved by selecting the corresponding entry in the Session Overview Area of the NightTrace Main Window (see “Session Overview Area” on page 5-39) and pressing the **Save Data Segment...** button (see “Save Data Segment...” on page 5-40).

## Zoom To Region

Sets the interval to the region between the mark and the current time line (inclusive).

### NOTE

See “Mark” on page 9-33 for more information on specifying a region.

When this action is performed:

- **Start Time** is set to the value of either the mark or the current time line, whichever is leftmost
- **End Time** is set to the value of either either the mark or the current time line, whichever is rightmost
- **Current Time** is set to the midpoint of the new interval; the current time line is positioned appropriately

## Zoom In

Accelerator: down-arrow

Each time the Zoom In button is pressed, the **Time Length** is reduced by the value of **Zoom Factor** thereby providing a more detailed view of a smaller interval. The interval is centered around the current time line.

The values of **Start Time**, **End Time**, **Start Event**, **End Event**, and **Event Count** are changed accordingly. (See “Interval Control Area” on page 9-37 for more information about these fields.)

## Zoom Out

Accelerator: up-arrow

Each time the Zoom Out button is pressed, the **Time Length** is multiplied by the value of **Zoom Factor** thereby providing a higher-level view of a larger interval. The interval is centered around the current time line.

The values of **Start Time**, **End Time**, **Start Event**, **End Event**, and **Event Count** are changed accordingly. (See “Interval Control Area” on page 9-37 for more information about these fields.)

## Refresh

Accelerator: Alt+R

Updates the grid to reflect the result of changes in configuration (see “Grid” on page 9-28).

Should be used when:

- opening a display page
- switching from *edit mode* to *view mode* (see “Mode Button” on page 9-36)
- changing a configuration parameter from view mode
- resizing the grid

## Mode Button

The **Mode** button appears to the right of the interval push buttons (see “Interval Push Buttons” on page 9-31) and toggles the mode in which display pages can be operated.



**Figure 9-24. Mode Button**

The icon to the left of the **Mode** button indicates the current mode:



indicates the display page is in *view mode*



indicates the display page is in *edit mode*

---

Placing the display page in *view mode* allows the user to view the trace data and perform operations related to the trace event data such as searching for points of interest or summarizing statistical information.

Placing the display page in *edit mode* allows the user to create, edit, or configure display objects on a display page (see “Operations on Display Objects” on page 10-12).

## Interval Control Area

The interval control area is a region of the display page that contains nine fields of statistics. If you have not already positioned your interval somewhere else, NightTrace displays in the interval control area the earliest 5 percent of your trace session. Usually this information is uninteresting and you want to see other parts of your trace session. You can do two things with the statistics in the interval control area:

- Read the fields to obtain information about the interval
- Edit the fields to change the interval

The screenshot shows a control panel with the following fields and values:

Start Time: 0,000000000s	Time Length: 2,190057076s	End Time: 2,190057076s	Current Time: 0,332708153s
Start Event: 0	Event Count: 2127	End Event: 2126	Apply Reset

**Figure 9-25. Interval Control Area**

All field values in the interval control area are non-negative numbers. Some fields have default values. Time fields all display the time in seconds with the “s” suffix. A description of each field follows. In the following text, *interval* is the time from **Start Time** through **End Time**.

### Start Time

The beginning time of the interval in seconds.

A valid change keeps **Start Time** less than the ending time in the trace session. The new interval starts at the specified time. **Time Length** remains unchanged, but other fields, including **End Time**, change appropriately.

If you set **Start Time** to the word `start`, NightTrace resets **Start Time** to the start time (0 microseconds) of the trace session.

### Start Event

The ordinal number (offset), not the trace event ID, of the first trace event in this interval.

A valid change keeps **Start Event** less than the number of trace events logged in the trace session. The new interval starts at the specified ordinal trace event number (offset). **Time Length** remains unchanged, but other fields change appropriately.

If you set **Start Event** to the word `start`, NightTrace resets **Start Event** to 0 and **Start Time** to 0 microseconds.

### Time Length

The amount of time between **Start Time** and **End Time**. Also known as the *interval*.

A valid change keeps **Time Length** greater than 0 and less than or equal to the last recorded time in the trace session. The new interval length is the specified length. **End Time** and other fields change appropriately.

If you set **Time Length** to the word `all` or an arbitrarily large number, NightTrace resets **Time Length** to the last time recorded in the trace event file(s) and changes other fields appropriately.

### Event Count

The quantity of trace events present in this interval. It is the difference between **End Event** and **Start Event** plus one.

A valid change keeps **Event Count** less than or equal to the ordinal position (offset) of the last trace event recorded in the trace session. The new trace event count is the specified count. Fields change appropriately.

If you set **Event Count** to the word `all` or an arbitrarily large number, NightTrace resets **Event Count** to the total number of trace events in your trace event file(s) and changes other fields appropriately.

### End Time

The ending time of the interval in seconds.

A valid change keeps **End Time** greater than the beginning time in the trace session and greater than or equal to **Time Length**. The new interval ends at the specified time. **Time Length** remains unchanged, but other fields, including **Start Time**, change appropriately.

If you change **End Time** so it is smaller than **Time Length**, NightTrace sets **End Time** to **Time Length**. If you set **End Time** to the word `end` or an arbitrarily large number, NightTrace resets **End Time** to the last time recorded in the trace event file(s) and changes other fields appropriately.

### End Event

The ordinal number (offset), not the trace event ID, of the last trace event in this interval.

A valid change keeps **End Event** non-negative. The new interval ends at the specified ordinal trace event number (offset). **Time Length** remains unchanged, but other fields change appropriately.

If you set **End Event** to the word `end`, or an arbitrarily large number, NightTrace resets **End Event** to the total number of trace events in your trace event file(s).

### Current Time

The present time within the interval in seconds.

If the new current time is *inside* the current interval, the current time line moves appropriately in any Columns (see "Column" on page 10-6) and the current interval remains unchanged.

If the new current time is *outside* the current interval, the interval shifts so the current time is centered in the interval, the current time line is centered in any Columns, and the interval length remains unchanged.

**Apply**

Validates any field change(s) in the interval control area (see “Interval Control Area” on page 9-37) and makes corresponding changes to other field(s), updates display objects on the grid (see “Grid” on page 9-28), and positions the current time line appropriately.

**Reset**

Restores changed field(s) in the interval control area (see “Interval Control Area” on page 9-37) to the value(s) they had the last time changes were applied.





Types of Display Objects . . . . .	10-3
Grid Label . . . . .	10-4
Data Box . . . . .	10-5
Column . . . . .	10-6
Event Graph . . . . .	10-6
State Graph . . . . .	10-7
Data Graph . . . . .	10-8
Ruler . . . . .	10-10
Operations on Display Objects . . . . .	10-12
Creating Display Objects . . . . .	10-12
Selecting Display Objects . . . . .	10-13
Moving Display Objects . . . . .	10-14
Resizing Display Objects . . . . .	10-14
Configuring Display Objects . . . . .	10-15
Grid Label . . . . .	10-17
Data Box . . . . .	10-19
Event Graph . . . . .	10-26
State Graph . . . . .	10-32
Data Graph . . . . .	10-39
Ruler . . . . .	10-47
Configuration Dialog Push Buttons . . . . .	10-48
Common Configuration Parameters . . . . .	10-50
Font . . . . .	10-50
Events . . . . .	10-50
Condition . . . . .	10-51
Processes . . . . .	10-51
Threads . . . . .	10-52
Nodes . . . . .	10-53
Horizontal Alignment . . . . .	10-53
Vertical Alignment . . . . .	10-54
Color Selection . . . . .	10-54
Font Selection . . . . .	10-55



## Display Objects

A display page contains *display objects* which filter, process, and display information based upon trace event data. These display objects are created and viewed on the display page.

Display objects, which are created via the **Create** menu (see “Create” on page 9-19) on the display page, can be thought of as combination filters and formatters for the trace event data. Every time a display object is updated, it filters through the trace data. The display object accepts input in the form of a trace event record, processes and reformats the information, and displays it.

The following information is in a trace event record:

- numeric trace event ID
- global process identifier (PID)
- NightTrace thread identifier (TID)
- time
- ordinal number (offset)
- optional arguments

You can use NightTrace functions to express any of these values (see “Functions” on page 11-4).

Although trace event data contains simple events, it implicitly contains states. The concepts of trace events and states are key to understanding display objects.

*trace event*                      Corresponds to the point in the execution of your application when a `trace_event()` call was executed. All the data logged at that time (trace event ID, arguments, etc.) is considered a trace event.

*state*                                A state is bounded by two trace events, a *start event* and an *end event*. An instance of a state is the period of time between the start event and end event, including the start and end events themselves. Additional conditions may be specified in a state definition to further constrain the state. Instances of individual states do not nest; that is, once a state becomes active, events that might normally satisfy the conditions for the start event are ignored until the end event is encountered.

Different types of display objects display information in different ways. Depending on the type of information you want to display, you choose the display object or objects you wish to create. You can then configure those display objects to filter out unwanted data and display the information that you want.

All display objects are rectangular with user-specified dimensions and have the following properties:

- Display objects can be dynamic or static. *Dynamic* means the contents vary depending on values in the trace event file and may change depending on the *current trace event*. *Static* means the contents do not change. All display objects except Grid Labels are dynamic (see “Grid Label” on page 10-4).
- Display objects can be textual or graphical. *Textual* means the contents consist of words or numbers. *Graphical* means the contents are lines or shapes, like a bar chart.
- Display objects can be scrollable or non-scrollable. *Scrollable* means the display object acts as a movable window into the trace event file.

## Types of Display Objects

The basic types of display objects are listed below and are discussed in the following sections.

- Grid Label

Static textual display object that contains a user-specified string of text and is used to label other display objects for clarity.

See “Grid Label” on page 10-4 for more information.

- Data Box

Dynamic display object that displays textual or numeric information related to a trace event or state attribute associated with the current time line. The main use of a Data Box is to display data that is variable in nature and does not lend itself to graphical representation.

See “Data Box” on page 10-5 for more information.

- Column

Dynamic display object that does not display data itself but holds the scrollable graphical display objects: State Graphs, Event Graphs, Data Graphs, and Rulers. Its purpose is to group together related graphical display objects.

See “Column” on page 10-6 for more information.

- Event Graph

Dynamic, scrollable display object that graphically displays trace events as vertical lines in a Column and indicates relative chronological positions of trace events since the trace started.

See “Event Graph” on page 10-6 for more information.

- State Graph

Dynamic, scrollable display object that graphically displays states as bars and other trace events as vertical lines in a Column and indicates relative chronological positions of trace events and states since the trace started. This display object is usually used if you want to know when the application enters and exits a particular user-defined state.

See “State Graph” on page 10-7 for more information.

- Data Graph

Dynamic, scrollable display object that graphically displays numeric values as vertical lines or bars in a Column and indicates the relative chronological position of the associated trace event. The height of the line or bar is proportional to the value and is scaled according to the minimum and maximum graph values specified. This dis-

play object is commonly used to display relative values of arguments in the trace event record.

See “Data Graph” on page 10-8 for more information.

- Ruler

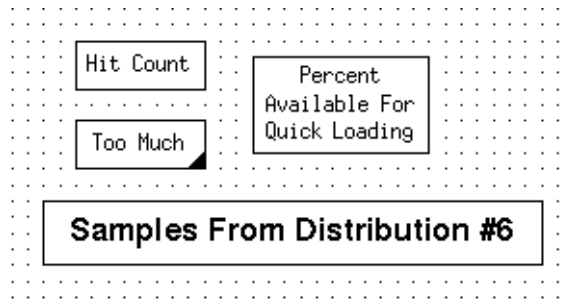
Static, scrollable display object resembling a ruler that graphically displays the time. Rulers are used in a Column with State Graphs, Event Graphs, and Data Graphs to show what time a trace event occurred.

See “Ruler” on page 10-10 for more information.

Each display page can hold multiple instances of these display objects, usually with each display object uniquely configured. All display objects on all display pages reflect the same interval and current time line; display object type, size, configuration, and position have no bearing.

## Grid Label

A *Grid Label* is a rectangle that contains a string of text. This text usually is a title or description of an adjacent display object on the grid and makes the display page easier to interpret. Grid Labels can appear anywhere on the grid, but they cannot go inside a Column. You can put several Grid Labels on a grid.



**Figure 10-1. Grid Label Examples**

Grid Labels are created by selecting the **Grid Label** menu item from the **Create** menu on the display page (see “Create” on page 9-19). See “Creating Display Objects” on page 10-12 for more information.

If the text is too long to fit into the Grid Label, the lower right corner of the box is filled in. If this occurs, you should resize the Grid Label. This is described in “Resizing Display Objects” on page 10-14. A newly created label contains the word `label`.

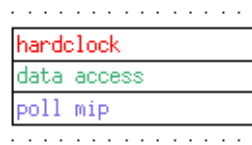
Grid Labels are static display objects. That is, a Grid Label does not change its appearance or contents depending on the trace event data.

In addition to specifying the text inside of the Grid Label, you also specify the color of the text (and background), the font of the text, and where in the box the text will appear (for example, top vs. bottom).

See “Grid Label” on page 10-17 for more information on configuring Grid Labels.

## Data Box

A *Data Box* is a rectangle that textually displays data from the trace event file. Although the data is usually related to the last trace event received, it can also be a cumulative total or other manipulations of data in the trace event file. Data Boxes are useful when you want to display data that does not lend itself to graphical representation, as shown in Figure 10-2. This figure shows three Data Boxes: the top Data Box contains the interrupt name, the middle contains the exception name and the bottom contains the syscall name.



**Figure 10-2. Data Box Examples**

Data Boxes are created by selecting the **Data Box** menu item from the **Create** menu on the display page (see “Create” on page 9-19). See “Creating Display Objects” on page 10-12 for more information.

If a value is too large to fit into the Data Box (e.g., a long trace event name), the lower right corner of the box is filled in. If this occurs, you should resize the Data Box (see “Resizing Display Objects” on page 10-14).

By default, numeric data is displayed in decimal integer. (For information about overriding this default, see “Event Map Files” on page 4-10, “format()” on page 11-110, and “get\_format()” on page 11-108.) A newly created Data Box contains a 0.

Data Boxes can appear anywhere on the grid except within a Column. You can put several Data Boxes on a grid.

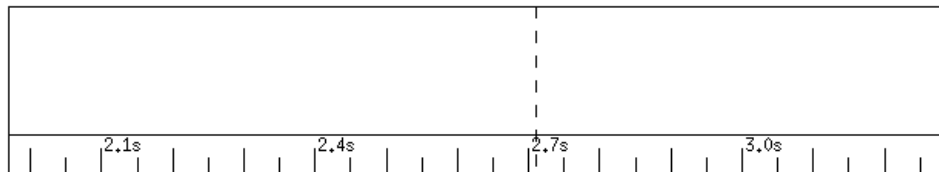
Some examples of data that you can configure a Data Box to show are:

- The name of the last trace event before the current time.
- The NightTrace thread name of the last trace event before the current time.
- A particular argument logged with the last trace event before the current time (See “arg()” on page 11-16.)
- The total amount of time the application was in a particular state before the current time (See “state\_dur()” on page 11-74 and “sum()” on page 11-100.)
- The number of times a particular trace event has occurred before the current time (See “event\_matches()” on page 11-36.)
- A string of characters generated by a format expression (See “format()” on page 11-110.)

See “Data Box” on page 10-19 for more information on configuring Data Boxes.

## Column

A Column holds State Graphs, Event Graphs, Data Graphs and Rulers. It provides a convenient way of associating these graphical display objects. Figure 10-3 shows a Column with a Ruler added to it.



**Figure 10-3. Column Example**

Columns are created by selecting the **Column** menu item from the **Create** menu on the display page (see “Create” on page 9-19). See “Creating Display Objects” on page 10-12 for more information.

When a *Column* is first created, it is an empty rectangle that does not display data of its own.

Columns ensure that all graphical display objects within them have the same physical starting point and ending point and the same time scale. Columns are not configured, so the only variations between Columns are in their height and width.

Without a Column, you cannot put any State Graphs, Event Graphs, Data Graphs or Rulers on your grid, so you must create a Column before you can create any of these display objects.

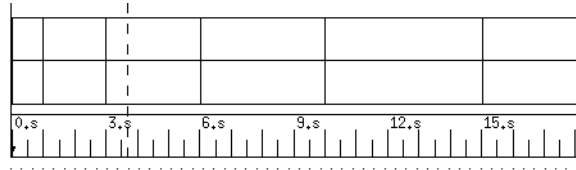
You can place a Column anywhere on the grid. You can put more than one Column on a grid. This allows you to group related graphical objects together. All of the Columns, however, show the same interval and current time in View mode.

To hold a Ruler and any other graphical display object, Columns must be at least five grid dots high. Wider Columns are recommended because they determine the resolution to which trace events can be displayed.

## Event Graph

An *Event Graph* represents trace events as a thin vertical line. Figure 10-4 shows an Event Graph with a Ruler below it.





**Figure 10-4. Event Graph Example**

Event Graphs are created by selecting the **Event Graph** menu item from the **Create** menu on the display page (see “Create” on page 9-19). See “Creating Display Objects” on page 10-12 for more information.

Event Graphs must be placed in a Column (see “Column” on page 10-6).

Some examples of information that an Event Graph can be used to display are:

- The times your application starts executing a particular subroutine
- The sequence of execution of various modules in your application
- The timing of the birth and death of child processes

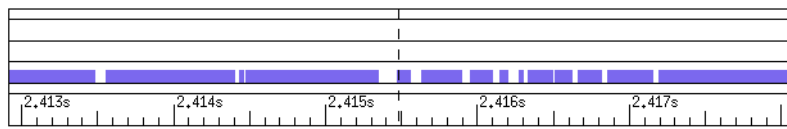
**NOTE**

In *view mode* (see “Mode Button” on page 9-36), to find out more information about a particular trace event, position the cursor on the line and click once with mouse button 2. Information about that trace event is displayed in the message display area.

See “Event Graph” on page 10-26 for more information on configuring Event Graphs.

## State Graph

A *State Graph* represents an instance of a state as a solid horizontal bar that starts when the state is active and ends when the state is inactive. A *state* is bounded by two user-specified trace events, a *start event* and an *end event*. A State Graph and a Ruler are shown in Figure 10-5.



**Figure 10-5. State Graph Example**

State Graphs are created by selecting the **State Graph** menu item from the **Create** menu on the display page (see “Create” on page 9-19). See “Creating Display Objects” on page 10-12 for more information.

State Graphs must be placed in a Column (see “Column” on page 10-6).

An instance of a state is the period of time between the start event and end event, including the start and end events themselves. Instances of the same state do not nest; thus, once a state becomes active, events that might normally satisfy the conditions for the start event are ignored until the end event is encountered.

A State Graph can display trace events in a manner identical to an Event Graph. This can be useful for saving screen space or detecting when state start and state end trace events occur out of order. For example, the trace event lines can show multiple state start trace events occurring before a state end trace event.

Some examples of information that State Graphs can be used to display are:

- The times your application is executing a particular subroutine
- The differences in the execution speed of parallel threads
- The time spent in contention for resources

#### **NOTE**

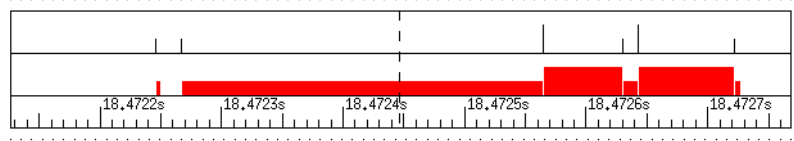
In *view mode* (see “Mode Button” on page 9-36), to find out more information about a particular trace event, position the cursor on a trace event line and click once with mouse button 2. Information about that trace event is displayed in the message display area. You can also click with mouse button 2 on the start and end of a displayed state to obtain information about the state start and state end trace events.

See “State Graph” on page 10-32 for more information on configuring State Graphs.

## **Data Graph**

A Data Graph represents data as either vertical lines or bars of varying height. The height of the line or bar is proportional to the value specified and is scaled according to the minimum and maximum graph values allowed. This display object is usually used to display values of arguments in the trace event record.

In Figure 10-6, the same set of data is used to draw two Data Graphs which differ only by the fill style. The top Data Graph uses vertical lines of varying height to represent the data. The bottom Data Graph uses solid bars of varying height; each bar extends to the next event recognized by the Data Graph.



**Figure 10-6. Data Graph Examples**

Data Graphs are created by selecting the **Data Graph** menu item from the **Create** menu on the display page (see “Create” on page 9-19). See “Creating Display Objects” on page 10-12 for more information.

Data Graphs must be placed in a Column (see “Column” on page 10-6).

Some examples of ways that a Data Graph can be used are:

- track the value of an expression over time
- identify when an application variable takes on an abnormally high or low value

When choosing a size for your Data Graphs, make sure that they are high enough for you to distinguish differences in data values.

#### **TIP**

The higher you make the Data Graph, the easier it is to differentiate similar data points.

#### **NOTE**

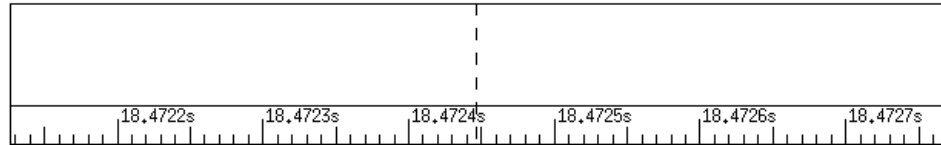
In *view mode* (see “Mode Button” on page 9-36), to find out about the trace event that caused the data value expression to be evaluated at a particular point, position the cursor on the line (or bar) and click once with mouse button 2. Information about the trace event is displayed in the message display area.

In *view mode*, to find out the value of a particular data item, position the cursor on the line (or bar) and click once with mouse button 3. The value of that data item is displayed in the message display area.

See “Data Graph” on page 10-39 for more information on configuring Data Graphs.

## Ruler

A *Ruler* graphically displays the time interval for the current dataset. Ruler display objects have major and minor hash marks to mark divisions of time since the first trace event was logged.





**Figure 10-7. Ruler Example**

Rulers are created by selecting the **Ruler** menu item from the **Create** menu on the display page (see “Create” on page 9-19). See “Creating Display Objects” on page 10-12 for more information.

Rulers must be placed in a Column (see “Column” on page 10-6) and should be at least three grid dots high.

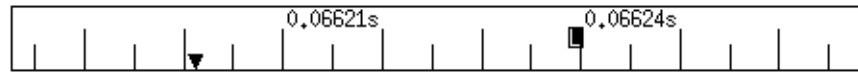
In addition to hash marks and numbers, other indicators that provide useful information about the trace data being displayed are:

---

D	a point in time where trace event had been discarded (see “Discard Events...” on page 9-34)
L	a point in time where NightTrace lost data (see “Preventing Trace Events Loss” on page A-1)
P	a point in time where the daemon logging trace data was paused
R	the point in time where the daemon logging trace data was resumed
?	a point in time where an erroneous timestamp was detected on a kernel trace data point
	a <i>mark</i> set by the user (see “Mark” on page 9-33)
	a <i>tag</i> set by the user (see “Tag” on page 9-32)

---

Figure 10-8 shows both a mark and a lost data indicator on a Ruler.



**Figure 10-8. Ruler Indicators**

By default, the indicators appear in reverse-video with the indicator displayed as white text over a colored background except for the mark which appears as a solid triangle. The colors of the various indicators as well as the foreground color and background color of the Ruler can be selected using the **Configure Ruler** dialog.

See “Ruler” on page 10-47 for more information on configuring Rulers.

## Operations on Display Objects

This section describes some operations you can perform on display objects. The operations discussed are:

- Creating display objects  
See “Creating Display Objects” on page 10-12.
- Selecting display objects  
See “Selecting Display Objects” on page 10-13.
- Moving display objects  
See “Moving Display Objects” on page 10-14.
- Resizing display objects  
See “Resizing Display Objects” on page 10-14.
- Configuring display objects  
See “Configuring Display Objects” on page 10-15.

### NOTE

The display page must be in *edit mode* in order to perform any of these operations on display objects. See “Mode Button” on page 9-36 for more information.

## Creating Display Objects

Creating display objects involves three steps: selecting the type of display object to be drawn, selecting the place on the grid where the display object will go, and selecting the size of the display object.

### NOTE

The display page must be in *edit mode* in order to create display objects. See “Mode Button” on page 9-36 for more information.

State Graphs, Event Graphs, Data Graphs and Rulers must be created inside a Column (see “Column” on page 10-6).

To create a display object and place it on the grid, do the following:

1. Select the type of display object you want to create from the **Create** menu (see “Create” on page 9-19) of the display page. (The mouse pointer changes to a crosshair).
2. Move the pointer until it is on the grid where you want to place a corner of the display object. As mentioned previously, some display objects go only inside of Columns. If the cursor is on the border of a Column or outside of one, you will not be able to draw these display objects. Note that the left and right sides of these display objects are determined by the Column, and you only have to place the pointer somewhere on the intended top or bottom edge of the display object.
3. Click and drag mouse button 1 until the display object is the size you want it to be. While you are sizing a display object, its boundaries are shown as dashed lines. Note that if you press the <ESC> key before releasing mouse button 1, the operation aborts. The display object is still loaded, as signified by the crosshair at the pointer location, so you can immediately try to recreate the display object. Also note that display objects must not overlap (except for graphical display objects, which must overlap a Column).
4. Release mouse button 1. The display object should appear on your grid with solid line boundaries, unless there was an error (e.g., you placed a Data Box on top of an existing Grid Label). Notice that the display object is also selected (corners have handles). This is in case you want to move, configure, or resize it at this time.

## Selecting Display Objects

Often, you must select a display object before performing grid and edit operations. For example, before you can resize a display object you must first select the display object.

### NOTE

The display page must be in *edit mode* in order to configure display objects. See “Mode Button” on page 9-36 for more information.

To select a single display object, simply click on the display object with mouse button 1. The display object now has handles at the corners, indicating that the display object is selected.

When display objects are inside a Column, it is sometimes difficult to select the Column. To select an unselected Column, hold down the <CONTROL> key and click mouse button 1. If you perform the same action in a selected Column, the Column is deselected.

You can select multiple display objects three different ways. The first way to select multiple display objects is as follows:

1. Position the cursor outside the display objects you want to select.

2. Click mouse button 1 and drag the mouse until the rectangle that is formed completely surrounds only the display objects you want to select. If a display object is not completely surrounded by the rectangle, it will not be selected.
3. Release mouse button 1. The display objects that were within the rectangle will now have handles at each corner.

The second way to select multiple display objects is by using the <Shift> key. Holding down the <Shift> key and clicking mouse button 1 while the cursor is in an unselected display object selects that display object without deselecting any other display objects. This allows you to select any set of display objects that you want. If you perform the same action in a display object that is already selected, the display object is deselected.

The third way to select multiple display objects is by using the **Select All** menu item on the **Edit** menu (see “Select All” on page 9-5).

## Moving Display Objects

To move a display object to somewhere else on the grid, do the following:

1. Select the display object(s). Refer to “Selecting Display Objects” on page 10-13.
2. Using the mouse button 2, click anywhere on or within the selected display object(s) and drag to the desired location.
3. Release the middle button.

### NOTE

The display page must be in *edit mode* in order to move display objects. See “Mode Button” on page 9-36 for more information.

When display objects are inside a Column, it is sometimes difficult to move the Column. To move a selected Column, hold down the <Control> key and click mouse button 2.

Display objects must not overlap, although certain display objects must be placed inside a Column. If you try to move a display object on top of another display object, NightTrace displays an error message in the message display area and aborts the move.

## Resizing Display Objects

To resize a display object on the grid, do the following:

1. Select the display object. See “Selecting Display Objects” on page 10-13 for more information.



2. Using mouse button 3, click on a handle and drag until the desired size is reached.
3. Release the right button.

#### NOTE

The display page must be in *edit mode* in order to resize display objects. See “Mode Button” on page 9-36 for more information.

When display objects are inside a Column, it is sometimes difficult to resize the Column. To resize a selected Column, hold down the <Control> key and click mouse button 3. Note that a Column cannot be vertically resized smaller than the minimum space required to hold all the State Graphs, Event Graphs, Data Graphs and Rulers that it contains.

Display objects must not overlap, although certain display objects must be placed inside a Column. If you try to resize a display object on top of another display object, NightTrace displays an error message in the message display area and aborts the resize.

## Configuring Display Objects

Double-clicking on a particular display object will bring up the configuration dialog for that display object. In addition, you may select the **Configure...** menu item from the **Edit** menu of any display page to bring up the configuration dialog for the selected display object (see “Configure...” on page 9-5).

#### NOTE

The display page must be in *edit mode* in order to configure display objects. See “Mode Button” on page 9-36 for more information.

The following sections discuss the configuration dialogs for each of the following display objects:

- Grid Label

See “Grid Label” on page 10-17.

- Data Box

See “Data Box” on page 10-19.

- Event Graph

See “Event Graph” on page 10-26.

- State Graph

See “State Graph” on page 10-32.

- Data Graph

See “Data Graph” on page 10-39.

- Ruler

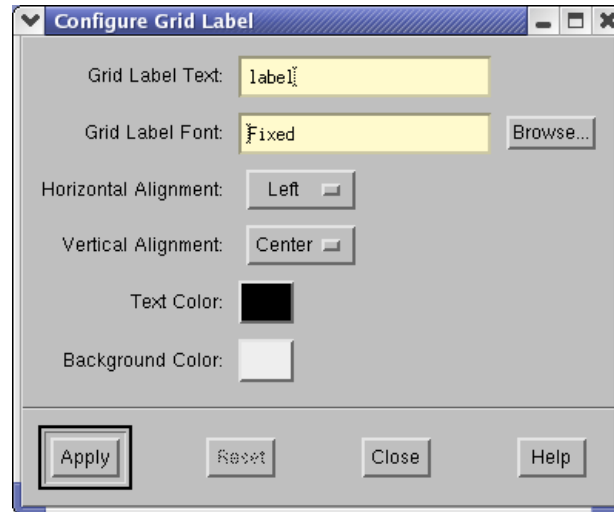
See “Ruler” on page 10-47.

See “Common Configuration Parameters” on page 10-50 for descriptions of the common configuration parameters that many of the display objects use.

## Grid Label

The Configure Grid Label dialog is shown in Figure 10-9.

See “Grid Label” on page 10-4 for more information.



**Figure 10-9. Configure Grid Label dialog**

### Grid Label Text

The text that is to be displayed in the Grid Label.

### Grid Label Font

The font in which the Grid Label Text is to be displayed.

See “Font” on page 10-50 for more information.

### Browse

Presents the Choose Font dialog (see “Font Selection” on page 10-55) allowing the user to specify a font by Family, Weight, Slant, and Size.

### Horizontal Alignment

Determines the justification of the text in the Grid Label.

See “Horizontal Alignment” on page 10-53.

### Vertical Alignment

Determines the vertical placement of the text in the Grid Label.

See “Vertical Alignment” on page 10-54.

**Text Color**

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the text displayed in the Grid Label. The **Text Color** should contrast well with the **Background Color** of the Grid Label.

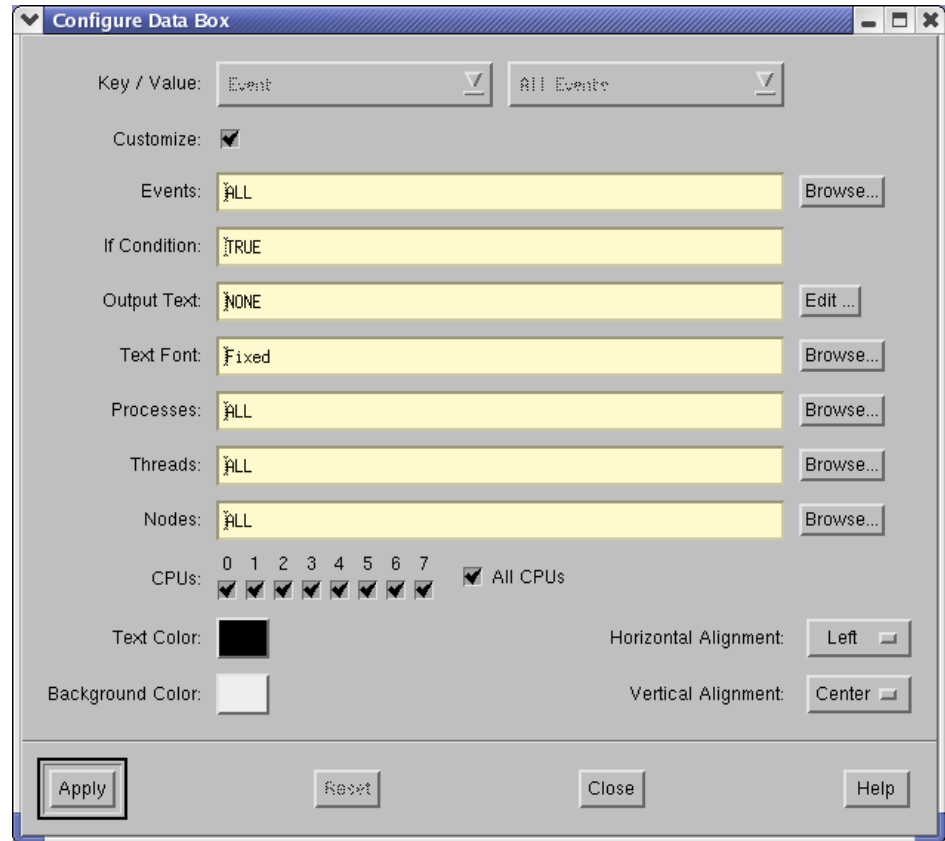
**Background Color**

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the background of the Grid Label. The **Background Color** should contrast well with the **Text Color**.

## Data Box

The Configure Data Box dialog is shown in Figure 10-10.

See “Data Box” on page 10-5 for more information.



**Figure 10-10. Configure Data Box dialog**

### Key / Value

These two drop-down menus are used in combination to provide a quick and efficient method for configuring a NightTrace Data Box.

The value chosen for the **Key** will determine the type of information displayed in the Data Box.

The **Value** drop-down provides a list of possible choices associated with the selected **Key**. The choices in this list are based on the trace dataset; for instance, if **Process ID** is selected for the **Key**, the **Value** drop-down will consist of those processes in the current dataset that logged trace events and/or those that were executing when kernel trace events were collected.

The Data Box can be further customized by selecting the **Customize** checkbox and modifying the remaining fields.

The **Key** drop-down contains the following items:

Event

information about the last occurrence of the event selected in the **Value** drop-down will be displayed in the Data Box

Exclude Event

information about the last occurrence of any trace event not including the event selected in the **Value** drop-down will be displayed in the Data Box

Tagged Event

information about the tagged event (see “Tag” on page 9-32) selected in the **Value** drop-down will be displayed in the Data Box

Predef. Qualified Event

information about the last occurrence of the previously-defined qualified event (see “Qualified Events” on page 11-113) specified in the **Value** drop-down will be displayed in the Data Box

Predef. Qualified State

information about the last occurrence of the previously-defined qualified state (see “Qualified States” on page 11-116) specified in the **Value** drop-down will be displayed in the Data Box

Process ID

information about the last occurrence of any trace event associated with the process specified in the **Value** drop-down will be displayed in the Data Box

Thread ID

information about the last occurrence of any trace event associated with the thread specified in the **Value** drop-down will be displayed in the Data Box

System Call

information about the last occurrence of the kernel starting or resuming execution of the particular system call selected in the **Value** drop-down will be displayed in the Data Box

System Call Leave

information about the last occurrence of the kernel exiting or suspending execution of the particular system call selected in the **Value** drop-down will be displayed in the Data Box

#### System Call Events

information about the last occurrence of the kernel starting, resuming, exiting, or suspending execution of the particular system call selected in the **Value** drop-down will be displayed in the Data Box

#### Interrupt

information about the last occurrence of the kernel starting execution of the particular interrupt selected in the **Value** drop-down will be displayed in the Data Box

#### Interrupt Leave

information about the last occurrence of the kernel exiting the particular interrupt selected in the **Value** drop-down will be displayed in the Data Box

#### Interrupt Events

information about the last occurrence of the kernel starting execution of or exiting the particular interrupt selected in the **Value** drop-down will be displayed in the Data Box

#### Exception

information about the last occurrence of the kernel starting or resuming execution of the particular exception selected in the **Value** drop-down will be displayed in the Data Box

#### Exception Leave

information about the last occurrence of the kernel exiting or suspending execution of the particular exception selected in the **Value** drop-down will be displayed in the Data Box

#### Exception Events

information about the last occurrence of the kernel starting, resuming, exiting, or suspending execution of the particular exception selected in the **Value** drop-down will be displayed in the Data Box

### Customize

Use this option to further configure your Data Box.

When **Customize** is checked, the **Key / Value** drop-down menus become disabled (desensitized). The remaining fields (which are populated with values corresponding to the **Key / Value** pair) may be modified to tailor the Data Box to your needs.

## NOTE

Once customization has occurred, unchecking the **Customize** button will result in all customized changes being discarded in favor of the displayed **Key / Value** selections.

## Events

Information about the trace events specified here will be displayed in the Data Box in the format specified by **Output Text**.

See “Events” on page 10-50 for more information.

## Browse

Presents a dialog allowing the user to select from a list of defined trace event names.

## NOTE

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

## If Condition

A boolean expression specifying criteria that must be met for information to be displayed in this Data Box.

See “Condition” on page 10-51 for more information.

## Output Text

The text that is to be displayed in the Data Box when all criteria specified by this configuration dialog is met.

This is usually specified using the `format()` command (see “format()” on page 11-110). For example:

```
format("%s event at offset %d", get_string(event, id), offset)
```

However, you may also enter a static string by placing double quotes around the desired text.

## Edit

Presents the **Edit Text** dialog in which to enter the output text. This dialog is useful when the desired text or `format()` string becomes too long to be easily edited directly in the **Output Text** field.



### **Text Font**

The font in which the **Output Text** is to be displayed.

See “Font” on page 10-50 for more information.

### **Browse**

Presents the **Choose Font** dialog (see “Font Selection” on page 10-55) allowing the user to specify a font by **Family**, **Weight**, **Slant**, and **Size**.

### **Processes**

Specify the processes to which this Data Box is restricted.

You may specify either PID values or the names of processes to which NightTrace has corresponding PID associations (e.g. from kernel trace data).

See “Processes” on page 10-51 for more information.

### **Browse**

Presents a dialog allowing the user to select from a list of the names of all processes that NightTrace discovers in the current trace data.

### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### **Threads**

Specify the threads to which this Data Box is restricted.

You may specify either TID values or the names of threads to which NightTrace has corresponding TID associations (e.g. from kernel trace data).

See “Threads” on page 10-52 for more information.

### **Browse**

Presents a dialog allowing the user to select from a list of the names of all threads that NightTrace discovers in the current user trace data.

### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

## Nodes

Specify the system node names to which this Data Box is restricted.

See “Nodes” on page 10-53 for more information.

### NOTE

The **Nodes** field is only meaningful for datasets captured from more than one system and is used to differentiate between them. Use of the *RCIM* timing source on daemon invocations is required for time synchronization in such cases. See the *Real-Time Clock and Interrupt Module User's Guide* (0891082) for more information.

## Browse

Presents a dialog allowing the user to select from a list of the names of all nodes that NightTrace discovers in the current trace data.

### NOTE

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

## CPUs

Specify which CPUs to which this Data Box is restricted.

### All CPUs

All CPUs are selected when this checkbox is checked.

## Text Color

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the text displayed in the Data Box. The **Text Color** should contrast well with the **Background Color** of the Data Box.

## Background Color

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the background of the Data Box. The **Background Color** should contrast well with the **Text Color**.

## Horizontal Alignment

Determines the justification of the text in the Data Box.

See “Horizontal Alignment” on page 10-53.

**Vertical Alignment**

Determines the vertical placement of the text in the Data Box.

See “Vertical Alignment” on page 10-54.

## Event Graph

The Configure Event Graph dialog is shown in Figure 10-11.

See “Event Graph” on page 10-6 for more information.

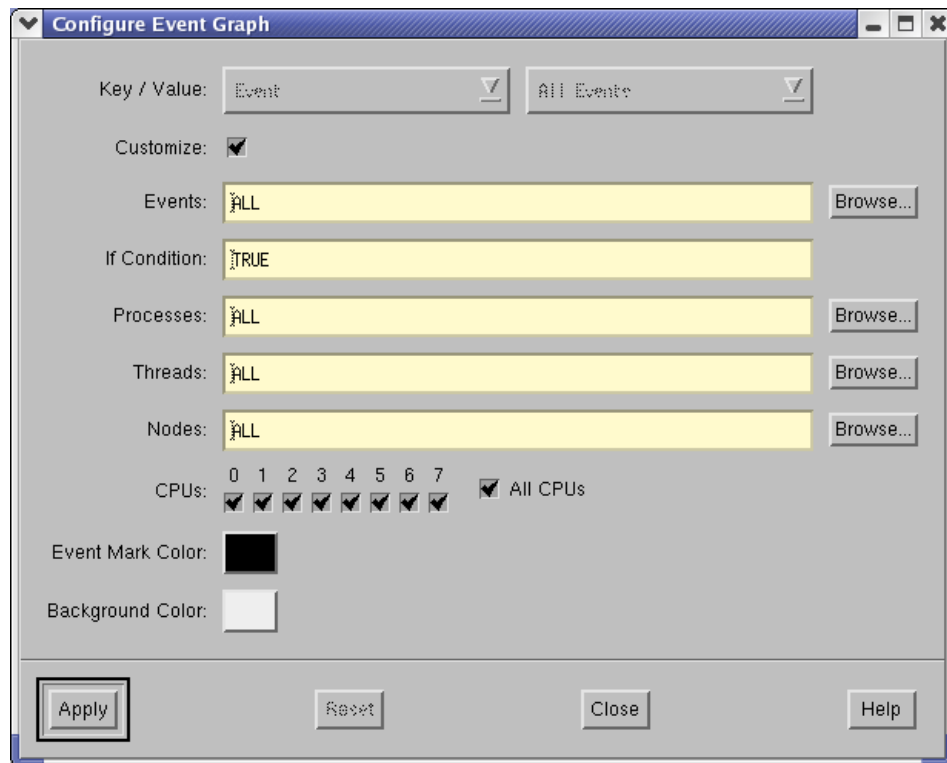


Figure 10-11. Configure Event Graph dialog

### Key / Value

These two drop-down menus are used in combination to provide a quick and efficient method for configuring a NightTrace Event Graph.

The value chosen for the **Key** will determine the events to be displayed on the Event Graph.

The **Value** drop-down provides a list of possible choices associated with the selected **Key**. The choices in this list are based on the trace dataset; for instance, if **Process ID** is selected for the **Key**, the **Value** drop-down will consist of those processes in the current dataset that logged trace events and/or those that were executing when kernel trace events were collected.

The Event Graph can be further customized by selecting the **Customize** checkbox and modifying the remaining fields.

The **Key** drop-down contains the following items:

Event

occurrences of the event selected in the **Value** drop-down will be displayed on the Event Graph

Exclude Event

occurrences of all events excluding the event selected in the **Value** drop-down will be displayed on the Event Graph

Tagged Event

occurrences of the tagged event (see “Tag” on page 9-32) selected in the **Value** drop-down will be displayed on the Event Graph

Predef. Qualified Event

occurrences of the previously-defined qualified event (see “Qualified Events” on page 11-113) specified in the **Value** drop-down will be displayed on the Event Graph

Predef. Qualified State

occurrences of the previously-defined qualified state (see “Qualified States” on page 11-116) specified in the **Value** drop-down will be displayed on the Event Graph

Process ID

occurrences of the events associated with the process specified in the **Value** drop-down will be displayed on the Event Graph

Thread ID

occurrences of the events associated with the thread specified in the **Value** drop-down will be displayed on the Event Graph

System Call

occurrences of those instances when the kernel starts or resumes execution of the particular system call selected in the **Value** drop-down will be displayed on the Event Graph

System Call Leave

occurrences of those instances when the kernel exits or suspends execution of the particular system call selected in the **Value** drop-down will be displayed on the Event Graph

System Call Events

occurrences of those instances when the kernel starts, resumes, exits, or suspends execution of the particular system call selected in the **Value** drop-down will be displayed on the Event Graph

#### Interrupt

occurrences of those instances when the kernel starts executing the particular interrupt selected in the **Value** drop-down will be displayed on the Event Graph

#### Interrupt Leave

occurrences of those instances when the kernel exits the particular interrupt selected in the **Value** drop-down will be displayed on the Event Graph

#### Interrupt Events

occurrences of those instances when the kernel starts executing or exits the particular interrupt selected in the **Value** drop-down will be displayed on the Event Graph

#### Exception

occurrences of those instances when the kernel starts or resumes execution of the particular exception selected in the **Value** drop-down will be displayed on the Event Graph

#### Exception Leave

occurrences of those instances when the kernel exits or suspends execution of the particular exception selected in the **Value** drop-down will be displayed on the Event Graph

#### Exception Events

occurrences of those instances when the kernel starts, resumes, exits, or suspends execution of the particular exception selected in the **Value** drop-down will be displayed on the Event Graph

### Customize

Use this option to further configure your Event Graph.

When **Customize** is checked, the **Key / Value** drop-down menus become disabled (desensitized). The remaining fields (which are populated with values corresponding to the **Key / Value** pair) may be modified to tailor the Event Graph to your needs.

#### NOTE

Once customization has occurred, unchecking the **Customize** button will result in all customized changes being discarded in favor of the displayed **Key / Value** selections.

### Events

The trace events to be displayed on this Event Graph.

See “Events” on page 10-50 for more information.

### **Browse**

Presents a dialog allowing the user to select from a list of defined trace event names.

### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### **If Condition**

A boolean expression specifying criteria that determines whether a trace event should be graphed. If the **If Condition** is true (and all other criteria are met), then a vertical line is drawn at the point in time that the trace event occurred.

See “Condition” on page 10-51 for more information.

### **Processes**

Specify the processes to which this Event Graph is restricted.

You may specify either PID values or the names of processes to which NightTrace has corresponding PID associations (e.g. from kernel trace data).

See “Processes” on page 10-51 for more information.

### **Browse**

Presents a dialog allowing the user to select from a list of the names of all processes that NightTrace discovers in the current trace data.

### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### **Threads**

Specify the threads to which this Event Graph is restricted.

You may specify either TID values or the names of threads to which NightTrace has corresponding TID associations (e.g. from kernel trace data).

See “Threads” on page 10-52 for more information.

### **Browse**

Presents a dialog allowing the user to select from a list of the names of all threads that NightTrace discovers in the current user trace data.

#### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### **Nodes**

Specify the system node names to which this Event Graph is restricted.

See “Nodes” on page 10-53 for more information.

#### **NOTE**

The **Nodes** field is only meaningful for datasets captured from more than one system and is used to differentiate between them. Use of the *RCIM* timing source on daemon invocations is required for time synchronization in such cases. See the *Real-Time Clock and Interrupt Module User's Guide* (0891082) for more information.

### **Browse**

Presents a dialog allowing the user to select from a list of the names of all nodes that NightTrace discovers in the current trace data.

#### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### **CPUs**

Specify which CPUs to which this Event Graph is restricted.

#### **All CPUs**

All CPUs are selected when this checkbox is checked.



**Event Mark Color**

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the vertical lines representing the trace events on the Event Graph. The **Event Mark Color** should contrast well with the **Background Color** of the Event Graph.

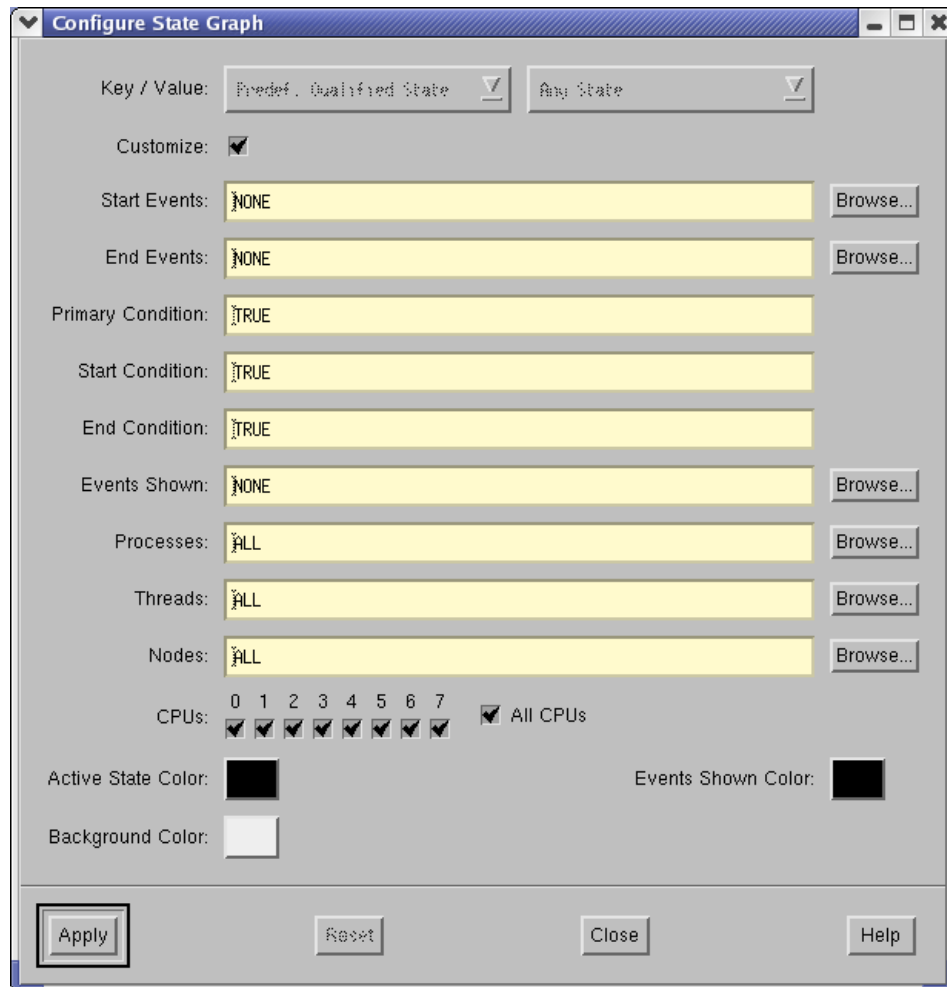
**Background Color**

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the background of the Event Graph. The **Background Color** should contrast well with the **Event Mark Color**.

## State Graph

The Configure State Graph dialog is shown in Figure 10-12.

See “State Graph” on page 10-7 for more information.



**Figure 10-12. Configure State Graph dialog**

A *state* is bounded by two user-specified trace events, a *start event* and an *end event*. An instance of a state is the period of time between the start event and end event, including the start and end events themselves. Instances of the same state do not nest; thus, once a state becomes active, events that might normally satisfy the conditions for the start event are ignored until the end event is encountered.

State Graphs indicate when a state is active by drawing a rectangle in the **Active State Color** that spans the time when the start state and end state criteria are met. In addition to drawing this state rectangle, State Graphs can behave exactly like Event Graphs by using the **Events Shown** and **Primary Condition** fields. Trace event lines are superim-

posed on the state rectangle, which is useful for diagnosing problems where the criteria for starting the state are met multiple times before the criteria for ending the state are met.

### Key / Value

These two drop-down menus are used in combination to provide a quick and efficient method for configuring a NightTrace State Graph.

The value chosen for the **Key** will determine the information displayed on the State Graph.

The **Value** drop-down provides a list of possible choices associated with the selected **Key**. The choices in this list are based on the trace dataset; for instance, if **Process ID** is selected for the **Key**, the **Value** drop-down will consist of those processes in the current dataset that logged trace events and/or those that were executing when kernel trace events were collected.

The State Graph can be further customized by selecting the **Customize** checkbox and modifying the remaining fields.

The **Key** drop-down contains the following items:

Predef. Qualified State

occurrences of the previously-defined qualified state (see “Qualified States” on page 11-116) specified in the **Value** drop-down will be displayed on the State Graph

System Call State

occurrences of the state whose *start events* are defined as those instances when the kernel starts or resumes execution of the particular system call selected in the **Value** drop-down and whose *end events* are comprised of those events when the kernel exits or suspends execution of that particular system call will be displayed on the State Graph

Interrupt State

occurrences of the state whose *start events* are defined as those instances when the kernel starts executing the particular interrupt selected in the **Value** drop-down and whose *end events* are comprised of those events when the kernel exits that particular interrupt will be displayed on the State Graph

Exception State

occurrences of the state whose *start events* are defined as those instances when the kernel starts or resumes execution of the particular exception selected in the **Value** drop-down and whose *end events* are comprised of those events when the kernel exits or suspends execution of that particular exception will be displayed on the State Graph

### Customize

Use this option to further configure your State Graph.

When **Customize** is checked, the **Key / Value** drop-down menus become disabled (desensitized). The remaining fields (which are populated with values corresponding to the **Key / Value** pair) may be modified to tailor the State Graph to your needs.

#### **NOTE**

Once customization has occurred, unchecking the **Customize** button will result in all customized changes being discarded in favor of the displayed **Key / Value** selections.

### **Start Events**

The **Start Events** parameter defines the trace events that can begin a state. Additional criteria that must be met for a state to begin may be specified in the form of an expression using the **Start Condition** parameter (see “Start Condition” on page 10-35).

The **Start Events** parameter works in combination with the **End Events** parameter which defines the trace events that can end a state (see “End Events” on page 10-34).

See “Events” on page 10-50 for more information.

#### **NOTE**

In order for a trace event to be considered a *start event*, all other criteria specified in this configuration dialog must be met: **Primary Condition**, **Start Condition**, **Processes**, **Threads**, **Nodes**, and **CPUs**.

### **Browse**

Presents a dialog allowing the user to select from a list of defined trace event names.

#### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### **End Events**

The **End Events** parameter defines the trace events that can end a state. Additional criteria that must be met for a state to end may be specified in the form of an expression using the **End Condition** parameter (see “End Condition” on page 10-36).

The **End Events** parameter works in combination with the **Start Events** parameter which defines the trace events that can begin a state (see “Start Events” on page 10-34).

See “Events” on page 10-50 for more information.

#### NOTE

In order for a trace event to be considered an *end event*, all other criteria specified in this configuration dialog must be met: **Primary Condition**, **End Condition**, **Processes**, **Threads**, **Nodes**, and **CPUs**.

#### Browse

Presents a dialog allowing the user to select from a list of defined trace event names.

#### NOTE

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

#### Primary Condition

A boolean expression specifying criteria that must be met as part of this State Graph.

If this condition is not met, the event is not regarded by the State Graph, even if the **Start Condition** or **End Condition** is true.

See “Condition” on page 10-51 for more information.

#### Start Condition

The **Start Condition** parameter is a boolean expression that allows the user to define criteria which must be met for a state to begin.

NightTrace evaluates this expression along with the **Start Events** parameter (see “Start Events” on page 10-34) and any other criteria specified in this dialog (e.g. **Primary Condition**, **Processes**, **CPUs**, etc.) to determine whether the state can begin.

**Start Condition** must not refer to its defining states. For example, it must not call `state_dur()`, `state_gap()`, `start` or `end` functions for these states. (See “Multi-State Functions” on page 11-73, “Start Functions” on page 11-37, and “End Functions” on page 11-55 for details.) Calling these functions for these states would be an attempt to define a state based on its own definition. Note that **Start Condition** may call all of these functions for qualified states.

See “Condition” on page 10-51 for more information.

## NOTE

Currently, NightTrace does not supported nesting of states. Thus, once the conditions which satisfy a *start event* are met, no other instances of that state can begin until the **End Condition** has been met.

### End Condition

The **End Condition** parameter is a boolean expression that allows the user to define criteria which must be met for a state to end.

NightTrace evaluates this expression along with the **End Events** parameter (see “End Events” on page 10-34) and any other criteria specified in this dialog (e.g. **Primary Condition**, **Processes**, **CPUs**, etc.) to determine whether the state can end.

**End Condition** must not refer to its defining states. For example, it must not call `state_dur()`, `state_gap()`, or `end` functions for these states. Calling these functions for these states would be an attempt to define a state based on its own definition. Note that **End Condition** may call start functions for these states because at this point in the state definition, the state has started. Note also that **End Condition** may call all of these functions for qualified states.

See “Condition” on page 10-51 for more information.

### Events Shown

The trace events to be displayed in this State Graph.

The color of the vertical lines that represent the **Events Shown** is determined by the value of the **Events Shown Color** as specified in this configuration dialog.

### Browse

Presents a dialog allowing the user to select from a list of defined trace event names.

## NOTE

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### Processes

Specify the processes to which this State Graph is restricted.

You may specify either PID values or the names of processes to which NightTrace has corresponding PID associations (e.g. from kernel trace data).

See “Processes” on page 10-51 for more information.

### **Browse**

Presents a dialog allowing the user to select from a list of the names of all processes that NightTrace discovers in the current trace data.

#### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### **Threads**

Specify the threads to which this State Graph is restricted.

You may specify either TID values or the names of threads to which NightTrace has corresponding TID associations (e.g. from kernel trace data).

See “Threads” on page 10-52 for more information.

### **Browse**

Presents a dialog allowing the user to select from a list of the names of all threads that NightTrace discovers in the current user trace data.

#### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### **Nodes**

Specify the system node names to which this State Graph is restricted.

See “Nodes” on page 10-53 for more information.

#### **NOTE**

The **Nodes** field is only meaningful for datasets captured from more than one system and is used to differentiate between them. Use of the *RCIM* timing source on daemon invocations is required for time synchronization in such cases. See the *Real-Time Clock and Interrupt Module User's Guide* (0891082) for more information.

### **Browse**

Presents a dialog allowing the user to select from a list of the names of all nodes that NightTrace discovers in the current trace data.

### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### **CPUs**

Specify which CPUs to which this State Graph is restricted.

#### **All CPUs**

All CPUs are selected when this checkbox is checked.

### **Active State Color**

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) to allow the user to specify the color of the solid horizontal bar that represents the instance of a state in the State Graph. The **Active State Color** should contrast well with the **Background Color** of the State Graph.

### **Events Shown Color**

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) to allow the user to specify the color of the vertical lines that represent the events specified by **Events Shown**. The **Events Shown Color** should contrast well with the **Background Color** of the State Graph.

### **Background Color**

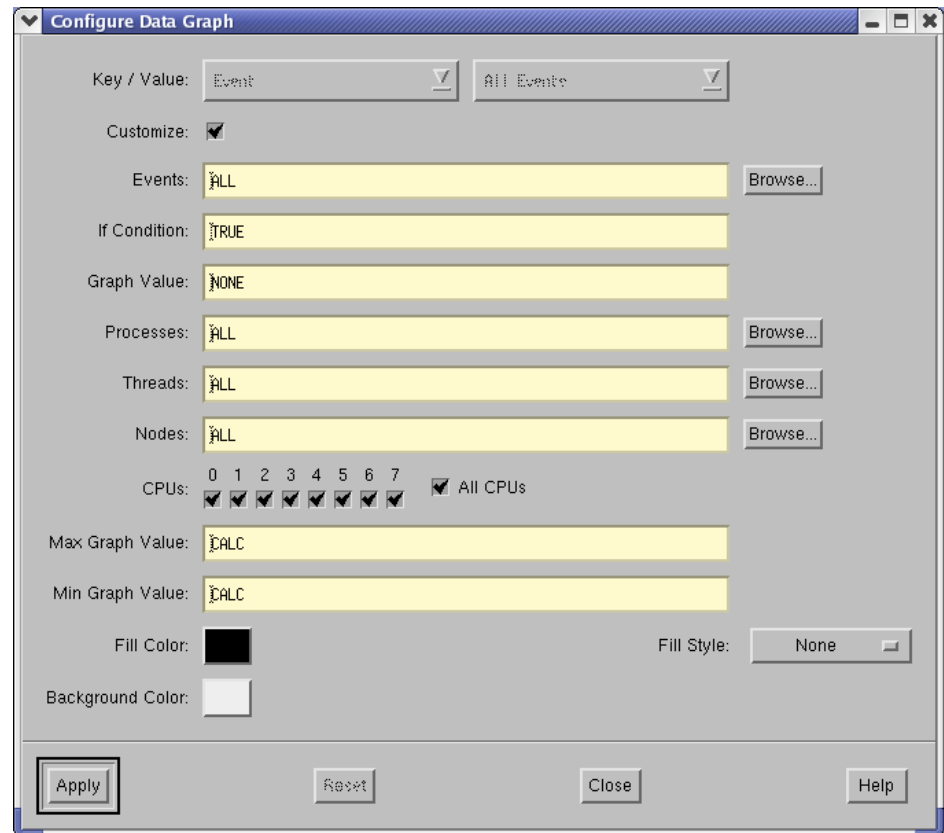
Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the background of the State Graph. The **Background Color** should contrast well with the **Active State Color** and the **Events Shown Color**.



## Data Graph

The Configure Data Graph dialog is shown in Figure 10-13.

See “Data Graph” on page 10-8 for more information.



**Figure 10-13. Configure Data Graph dialog**

### Key / Value

These two drop-down menus are used in combination to provide a quick and efficient method for configuring a NightTrace Data Graph.

The value chosen for the **Key** will determine the type of information displayed on the Data Graph.

The **Value** drop-down provides a list of possible choices associated with the selected **Key**. The choices in this list are based on the trace dataset; for instance, if **Process ID** is selected for the **Key**, the **Value** drop-down will consist of those processes in the current dataset that logged trace events and/or those that were executing when kernel trace events were collected.

The Data Graph can be further customized by selecting the **Customize** checkbox and modifying the remaining fields.

The **Key** drop-down contains the following items:

Event

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the event selected in the **Value** drop-down

Exclude Event

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of any event excluding the event selected in the **Value** drop-down

Tagged Event

the data specified by the **Graph Value** will be displayed on the Data Graph for the occurrence of the tagged event (see “Tag” on page 9-32) selected in the **Value** drop-down

Predef. Qualified Event

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the previously-defined qualified event (see “Qualified Events” on page 11-113) specified in the **Value** drop-down

Predef. Qualified State

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the previously-defined qualified state (see “Qualified States” on page 11-116) specified in the **Value** drop-down

Process ID

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of any event associated with the process specified in the **Value** drop-down

Thread ID

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of any event associated with the thread specified in the **Value** drop-down

System Call

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the kernel starting or resuming execution of the particular system call selected in the **Value** drop-down

System Call Leave

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the kernel exiting or suspending execution of the particular system call selected in the **Value** drop-down

#### System Call Events

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the kernel starting, resuming, exiting, or suspending execution of the particular system call selected in the **Value** drop-down

#### Interrupt

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the kernel starting execution of the particular interrupt selected in the **Value** drop-down

#### Interrupt Leave

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the kernel exiting the particular interrupt selected in the **Value** drop-down

#### Interrupt Events

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the kernel starting execution of or exiting the particular interrupt selected in the **Value** drop-down

#### Exception

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the kernel starting or resuming execution of the particular exception selected in the **Value** drop-down

#### Exception Leave

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the kernel exiting or suspending execution of the particular exception selected in the **Value** drop-down

#### Exception Events

the data specified by the **Graph Value** will be displayed on the Data Graph for each occurrence of the kernel starting, resuming, exiting, or suspending execution of the particular exception selected in the **Value** drop-down

### **Customize**

Use this option to further configure your Data Graph.

When **Customize** is checked, the **Key / Value** drop-down menus become disabled (desensitized). The remaining fields (which are populated with values corresponding to the **Key / Value** pair) may be modified to tailor the Data Graph to your needs.

## NOTE

Once customization has occurred, unchecking the **Customize** button will result in all customized changes being discarded in favor of the displayed **Key / Value** selections.

## Events

The trace events to be displayed by this Data Graph.

Data related to the trace events specified here will be displayed in the Data Graph in the format specified by **Graph Value**.

See “Events” on page 10-50 for more information.

## Browse

Presents a dialog allowing the user to select from a list of defined trace event names.

## NOTE

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

## If Condition

A boolean expression specifying criteria that must be met for information to be displayed on this Data Graph.

See “Condition” on page 10-51 for more information.

## Graph Value

The value to be graphed on the Data Graph.

The **Graph Value** can be any value between **Min Graph Value** and **Max Graph Value** and is usually related to the event. For instance, to graph the value of the second argument in the trace event record meeting all criteria specified by this configuration dialog:

`arg2`

should be entered for **Graph Value**.

## Processes

Specify the processes to which this Data Graph is restricted.

You may specify either PID values or the names of processes to which NightTrace has corresponding PID associations (e.g. from kernel trace data).

See “Processes” on page 10-51 for more information.

### **Browse**

Presents a dialog allowing the user to select from a list of the names of all processes that NightTrace discovers in the current trace data.

### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### **Threads**

Specify the threads to which this Data Graph is restricted.

You may specify either TID values or the names of threads to which NightTrace has corresponding TID associations (e.g. from kernel trace data).

See “Threads” on page 10-52 for more information.

### **Browse**

Presents a dialog allowing the user to select from a list of the names of all threads that NightTrace discovers in the current user trace data.

### **NOTE**

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

### **Nodes**

Specify the system node names to which this Data Graph is restricted.

See “Nodes” on page 10-53 for more information.

## NOTE

The **Nodes** field is only meaningful for datasets captured from more than one system and is used to differentiate between them. Use of the *RCIM* timing source on daemon invocations is required for time synchronization in such cases. See the *Real-Time Clock and Interrupt Module User's Guide* (0891082) for more information.

## Browse

Presents a dialog allowing the user to select from a list of the names of all nodes that NightTrace discovers in the current trace data.

## NOTE

To select multiple items, press the **Ctrl** key while selecting individual items in the list or hold the **Shift** key to select a range of items.

## CPUs

Specify which CPUs to which this Data Graph is restricted.

### All CPUs

All CPUs are selected when this checkbox is checked.

## Max Graph Value

The **Max Graph Value** parameter determines what data value corresponds to the top of the Data Graph.

The possible values are integers or *CALC*. If an integer is specified as the maximum, any data that is equal to or greater than that value results in a line or bar that goes to the top of the Data Graph. If *CALC* is specified, the maximum value will be the greatest value found in the trace event run up to that point in time. Note that the maximum can change as time increases and new maximums are encountered.

## Min Graph Value

The **Min Graph Value** parameter determines what data value corresponds to the bottom of the Data Graph.

The possible values are integers or *CALC*. If an integer is specified as the minimum, any data that is equal to or less than that value will result in no line or bar on the Data Graph. If *CALC* is specified, the minimum value will be the smallest value found in the trace event run up to that point in time. Note that the minimum can change as time increases and new minimums are encountered.

## Fill Color

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) to allow the user to specify the color of the vertical line or solid horizontal bar that represents the trace event in the Data Graph when either **None** or **Solid** is selected for the **Fill Style**. The **Fill Color** should contrast well with the **Background Color** of the Data Graph.

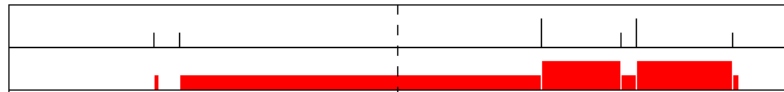
## Fill Style

The **Fill Style** parameter determines the style of Data Graph created.

The possible choices are:

<b>None</b>	a vertical line is drawn only at the time of a trace event
<b>Solid</b>	all space to the right of a trace event will be filled in the color specified by <b>Fill Color</b> until the next trace event is encountered
<b>Solid by Value</b>	all space to the right of a trace event will be filled in a color unique to the value being shown

Figure 10-14 shows the difference between **Solid** and **None**.



**Figure 10-14. Fill Style - Solid vs. None**

## Background Color

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the background of the Data Graph. The **Background Color** should contrast well with the **Fill Color**.

Figure 10-15 shows the same set of data drawn in three Data Graphs, each configured differently. The data range in value from 1 to 6 and are shown at the bottom of the figure.

- The top Data Graph is configured with a minimum of 2 and a maximum of 4. Notice that several bars reach the top of the Data Graph even though they represent different data values; also note that there is no bar where data has a value less than the minimum.
- The middle Data Graph is configured with a minimum of 0 and a maximum of 10. Notice that the bars do not reach the top of the Data Graph and that the differences between values are harder to discern.
- The bottom Data Graph is configured with a minimum of 0 and a maximum set to **CALC**. Notice that the two occurrences of the maximum value of six cause bars to reach the top of the Data Graph.

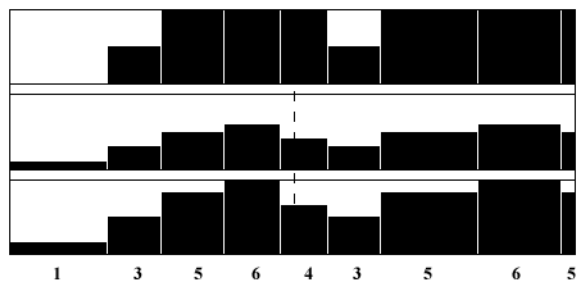


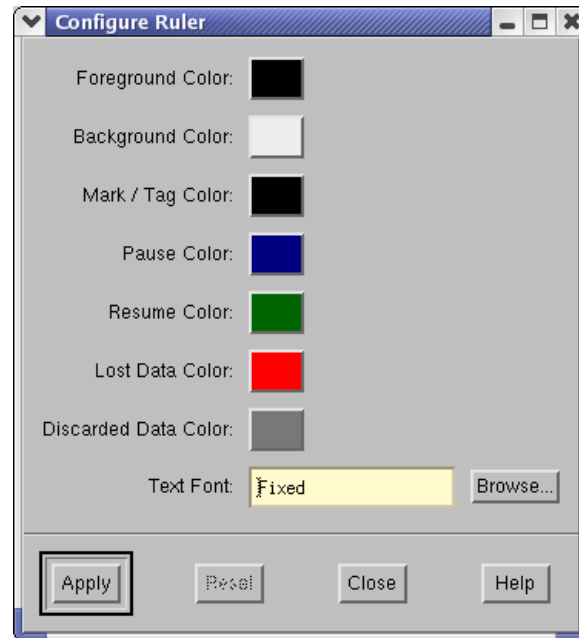
Figure 10-15. Maximum vs. Minimum Values



## Ruler

The Configure Ruler dialog is shown in Figure 10-16.

See “Ruler” on page 10-10 for more information.



**Figure 10-16. Configure Ruler dialog**

### Foreground Color

Presents the Choose Color dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the Ruler markings. The Foreground Color should contrast well with the Background Color.

### Background Color

Presents the Choose Color dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the background of the Ruler. The Background Color should contrast well with the other items displayed on the Ruler.

### Mark / Tag Color

Presents the Choose Color dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of both the *mark* and *tag* indicators which appear on the Ruler. The Mark / Tag Color should contrast well with the Background Color.

Figure 10-17 shows both a mark and a tag on a Ruler



**Figure 10-17. Mark and Tag Indicators**

See “Mark” on page 9-33 and “Tag” on page 9-32 for more information about these indicators.

**Pause Color**

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the reverse-video “P” indicator used to show the point in time where the daemon logging trace data was paused.

**Resume Color**

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the reverse-video “R” indicator used to show the point in time where the daemon logging trace data was resumed.

**Lost Data Color**

The **Lost Data Color** specifies the color of the reverse-video “L” that is placed on a Ruler where NightTrace lost data.

See “Preventing Trace Events Loss” on page A-1 for more information on lost data.

**Discarded Data Color**

Presents the **Choose Color** dialog (see “Color Selection” on page 10-54) allowing the user to specify the color of the reverse-video “D” indicator used to show where trace events have been discarded.

**Text Font**

The font in which the numeric values on the Ruler are displayed.

See “Font” on page 10-50 for more information.

**Browse**

Presents the **Choose Font** dialog (see “Font Selection” on page 10-55) allowing the user to specify a font by **Family**, **Weight**, **Slant**, and **Size**.

**Configuration Dialog Push Buttons**

The following push buttons appear on all display object configuration dialogs.

**Apply**

Validate the changes you made to the configuration parameters, and apply the changes to the display object.

**Reset**

Discard all changes made since the last Apply.

**Close**

Discard any changes made since the last change was applied and close the window.

**Help**

Display the help topic for the display object configuration dialog.

## Common Configuration Parameters

Many of the display objects share common configuration parameters. These include::

- Font (see “Font” on page 10-50)
- Events (see “Events” on page 10-50)
- Condition (see “Condition” on page 10-51)
- Processes (see “Processes” on page 10-51)
- Threads (see “Threads” on page 10-52)
- Nodes (see “Nodes” on page 10-53)
- Horizontal Alignment (see “Horizontal Alignment” on page 10-53)
- Vertical Alignment (see “Vertical Alignment” on page 10-54)

For each configuration parameter that pertains to color, there is an equivalent X resource. See Appendix B for more information.

### Font

The font description is specified as a Unix-centric font name of the form:

*-foundry-family-weight-slant-setwidth-addstyle-pixel-point-resx-resy-spacing-width-charset-encoding*

#### NOTE

The ‘\*’ character may be used to skip individual fields that the user does not care about. There must be exactly one ‘\*’ for each field skipped; a ‘\*’ at the end of the font description skips any remaining fields.

For instance, to specify a 12-point bold Helvetica font with no slant, the font description would be:

**\*-helvetica-bold-r-normal-\*-12-\***

The Choose Font dialog (see “Font Selection” on page 10-55) allows the user to specify a font by specifying Family, Weight, Slant, and Size.

### Events

The Events field restricts the trace events on which the display object can display information. The display object ignores any trace event IDs or trace event names that are not specified by this field. If an explicit list of trace event names and trace event IDs is specified, the names and IDs on the list must be separated by commas. Only listed trace events are examined.

The **Events** field can contain any meaningful combination of the following:

ALL	All trace events
ALLUSER	All user trace events
ALLKERNEL	All kernel trace events
NONE	No trace events
0, 1, 2, ..., 4095	Listed user trace event IDs
4100, 4101, 4102, ..., 4300	Listed kernel trace events IDs
A comma-separated list of alphanumeric strings beginning with letters (underscores are allowed; spaces are not allowed)	Trace event names as specified in an event map file (see “Event Map Files” on page 4-10 for more information)

## Condition

Conditions are boolean expressions which specify criteria that must be met as part of the display object configuration. (See “Expressions” on page 11-1 for more information.)

Conditions are boolean, i.e., they should evaluate to `false` (0) or `true` (non-zero). If the condition evaluates to true, then the appropriate information is displayed in the display object (assuming all other criteria are met).

Some examples of valid conditions and their effect on the display object are shown in Table 10-1.

**Table 10-1. Examples of Conditions**

Condition	Effect
<code>true</code>	Always evaluated
<code>false</code>	Never evaluated
<code>id() == 200</code>	Evaluated if current trace event ID is equal to 200
<code>id() &lt; 200</code>	Evaluated if current trace event ID is less than 200
<code>pid() == 237</code>	Evaluated if current global process ID is equal to 237
<code>tid() == 895</code>	Evaluated if current NightTrace thread ID is equal to 895
<code>cpu() == 2    cpu() == 4</code>	Evaluated if current trace event occurred on CPU 2 or 4

## Processes

On Linux systems, a *global process identifier* (PID) is the operating system process identifier.

On PowerMAX OS systems, a *global process identifier* (PID) is a 32-bit integer which includes a 16-bit integer *raw PID* and a 16-bit integer *lightweight process identifier* (LWPID). The syntax for specifying a PID on a PowerMAX OS system is:

*raw\_PID' LWPID*

The **Processes** field contains the list of global process identifiers (PIDs) or process names from which the display object will accept trace events. If the trace event did not occur in a process listed in this parameter, the trace event is ignored. If a number or name is specified that is not a valid PID, a warning message is displayed. Multiple numbers and names must be separated by commas.

The **Processes** field can be any meaningful combination of the following:

ALL	All PIDs
NONE	No PIDs
123, 456, 789, ...	Listed PIDs
A comma-separated list of alphanumeric strings beginning with letters (underscores are allowed; spaces are not allowed)	Process names

## Threads

A *NightTrace thread identifier* (TID) is a 32-bit integer. It includes a 16-bit integer *raw PID* and a 16-bit integer *C thread* or *Ada task identifier*. If neither C threads nor Ada tasks are in use, then the 16-bit integer will always be zero. The syntax for specifying a TID is:

*raw\_PID' task\_id*

or:

*raw\_PID' thread\_id*

The **Threads** field contains the list of NightTrace thread identifiers (TIDs) or thread names from which the display object will accept trace events. If the trace event did not occur in a thread listed in this parameter, the trace event is ignored. If a number or name is specified that is not a valid TID, a warning message is displayed. Multiple numbers and names must be separated by commas.

The **Threads** field can be any meaningful combination of the following:

ALL	All TIDs
NONE	No TIDs
123'1, 456'1, 789'1, ...	Listed TIDs
A comma-separated list of alphanumeric strings beginning with letters (underscores are allowed; spaces are not allowed)	The name of a thread as specified in the <code>trace_open_thread()</code> call (see "trace_open_thread()" on page 2-10 for more information)

## Nodes

When NightTrace processes a trace file which was timestamped by an RCIM synchronized tick clock, it internally assigns a node identifier to each node/host name represented by a trace file.

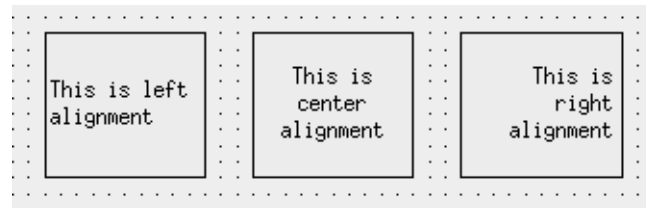
The **Nodes** field contains the list of node identifiers or node names from which the display object will accept trace events. If the trace event did not occur on a node listed in this parameter, the trace event is ignored. If a number or name is specified that is not a valid node, a warning message is displayed. Multiple numbers and names must be separated by commas.

The **Nodes** field can be any meaningful combination of the following:

ALL	All nodes
NONE	No nodes
0, 1, 4	Listed node IDs
A comma-separated list of host names (spaces are not allowed)	The name of a node/host

## Horizontal Alignment

The **Horizontal Alignment** parameter determines the justification of the text in the display object. Figure 10-18 shows what each type of horizontal alignment looks like.



**Figure 10-18. Horizontal Alignment**

### Left

Text is justified on the left side of the display object.

### Center

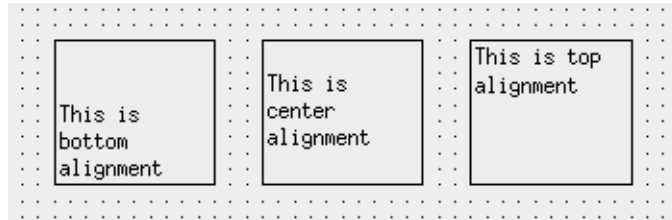
Text is horizontally centered in the display object.

### Right

Text is justified on the right side of the display object.

## Vertical Alignment

The **Vertical Alignment** determines whether text in the object will float to the top, be placed in the middle, or sink to the bottom of the display object. Figure 10-19 shows what each type of vertical alignment looks like in a Grid Label object (see “Grid Label” on page 10-4).



**Figure 10-19. Vertical Alignment**

### Center

Text is vertically centered in the display object.

### Bottom

Text sinks to the bottom of the display object.

### Top

Text floats to the top of the display object.

## Color Selection

The **Color Selection** dialog allows the user to specify the color of the various attributes of display objects by selecting the desired **Red**, **Green**, and **Blue** values.



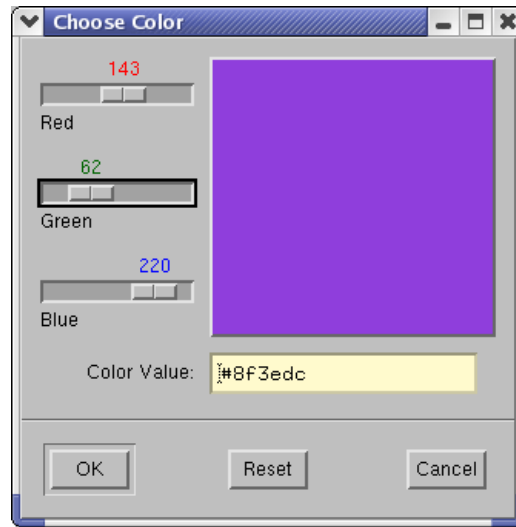


Figure 10-20. Choose Color dialog

### Color Value

In addition to using the Red, Green, and Blue sliders to select a color, you may also enter the name of any color found in `/usr/lib/X11/rgb.txt`.

## Font Selection

The Choose Font dialog allows the user to specify a font by selecting the desired Family, Weight, Slant, and Size.

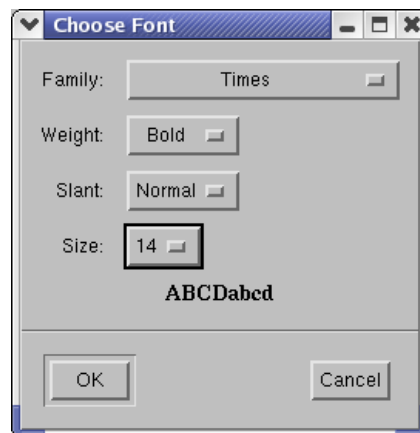


Figure 10-21. Choose Font dialog



Expressions . . . . .	11-1
Operators . . . . .	11-1
Operands . . . . .	11-2
Constants . . . . .	11-2
Functions . . . . .	11-4
Function Parameters . . . . .	11-7
Function Terminology . . . . .	11-8
Trace Event Functions . . . . .	11-14
id() . . . . .	11-15
arg() . . . . .	11-16
arg_dbl() . . . . .	11-17
num_args() . . . . .	11-18
pid() . . . . .	11-19
raw_pid() . . . . .	11-20
lwpid() . . . . .	11-21
thread_id() . . . . .	11-22
task_id() . . . . .	11-23
tid() . . . . .	11-24
cpu() . . . . .	11-25
offset() . . . . .	11-26
time() . . . . .	11-27
node_id() . . . . .	11-28
pid_table_name() . . . . .	11-29
tid_table_name() . . . . .	11-30
node_name() . . . . .	11-31
process_name() . . . . .	11-32
task_name() . . . . .	11-33
thread_name() . . . . .	11-34
Multi-Event Functions . . . . .	11-35
event_gap() . . . . .	11-35
event_matches() . . . . .	11-36
State Functions . . . . .	11-37
Start Functions . . . . .	11-37
start_id() . . . . .	11-38
start_arg() . . . . .	11-39
start_arg_dbl() . . . . .	11-40
start_num_args() . . . . .	11-41
start_pid() . . . . .	11-42
start_raw_pid() . . . . .	11-43
start_lwpid() . . . . .	11-44
start_thread_id() . . . . .	11-45
start_task_id() . . . . .	11-46
start_tid() . . . . .	11-47
start_cpu() . . . . .	11-48
start_offset() . . . . .	11-49
start_time() . . . . .	11-50
start_node_id() . . . . .	11-51

start_pid_table_name()	11-52
start_tid_table_name()	11-53
start_node_name()	11-54
End Functions	11-55
end_id()	11-56
end_arg()	11-57
end_arg_dbl()	11-58
end_num_args()	11-59
end_pid()	11-60
end_raw_pid()	11-61
end_lwpid()	11-62
end_thread_id()	11-63
end_task_id()	11-64
end_tid()	11-65
end_cpu()	11-66
end_offset()	11-67
end_time()	11-68
end_node_id()	11-69
end_pid_table_name()	11-70
end_tid_table_name()	11-71
end_node_name()	11-72
Multi-State Functions	11-73
state_gap()	11-73
state_dur()	11-74
state_matches()	11-75
state_status()	11-76
Offset Functions	11-77
offset_id()	11-78
offset_arg()	11-79
offset_arg_dbl()	11-80
offset_num_args()	11-81
offset_pid()	11-82
offset_raw_pid()	11-83
offset_lwpid()	11-84
offset_thread_id()	11-85
offset_task_id()	11-86
offset_tid()	11-87
offset_cpu()	11-88
offset_time()	11-89
offset_node_id()	11-90
offset_pid_table_name()	11-91
offset_tid_table_name()	11-92
offset_node_name()	11-93
offset_process_name()	11-94
offset_task_name()	11-95
offset_thread_name()	11-96
Summary Functions	11-97
min()	11-97
max()	11-98
avg()	11-99
sum()	11-
100	
min_offset()	11-
101	

max_offset() .....	11-
102	
summary_matches() .....	11-
103	
Format and Table Functions .....	11-
104	
get_string() .....	11-
104	
get_item() .....	11-
106	
get_format() .....	11-
108	
format() .....	11-
110	
Macros .....	11-
111	
Qualified Events .....	11-
113	
Qualified States .....	11-
116	
NightTrace Qualified Expressions .....	11-
119	
Edit NightTrace Qualified Expression .....	11-
122	



# Using Expressions

NightTrace allows you to define expressions in the form of macros, qualified events, and qualified states to aid in the analysis of trace data (see “Expressions” on page 11-1).

*Macros* are named expressions provided for flexibility and convenience (see “Macros” on page 11-111).

*Qualified events* provide a means for referencing a set of one or more trace events which may be restricted by conditions specified by the user (see “Qualified Events” on page 11-113).

*Qualified states* provide a means for defining regions of time based on specific starting and ending events and restricted by conditions specified by the user (see “Qualified States” on page 11-116).

NightTrace qualified expressions are created and configured using the **Edit NightTrace Qualified Expression** dialog (see “Edit NightTrace Qualified Expression” on page 11-122) and are managed using the **NightTrace Qualified Expressions** dialog (see “NightTrace Qualified Expressions” on page 11-119).

## Expressions

NightTrace expressions are comprised of a combination of *operators* and *operands* and can evaluate to numbers, strings, or boolean values.

See “Operators” on page 11-1 for a list of valid operators and “Operands” on page 11-2 for a discussion of valid operands.

## Operators

Operators in NightTrace expressions include:

- arithmetic operators: `()`, `*`, `/`, `%` (modulo), `+`, `-`, unary `-`
- shift operators: `<<`, `>>`
- bitwise operators: `~` (not), `&` (and), `^` (exclusive or), `|` (or)
- logical operators: `!` (not), `&&` (and), `||` (or)
- relational operators: `<`, `<=`, `>`, `>=`, `==` (equivalence), `!=` (non-equivalence)
- conditional operator: `expr ? true_value : false_value`
- unary casts to data types (where the parentheses are required): e.g., `(int)`

NightTrace operators follow the operator precedence rules of the C programming language.

## Operands

Operands include:

- constants (see “Constants” on page 11-2)
- macro calls (see “Macros” on page 11-111)
- function calls (see “Functions” on page 11-4)
- qualified events (*in functions only*) (see “Qualified Events” on page 11-113)
- qualified states (*in functions only*) (see “Qualified States” on page 11-116)

Operand types are largely based on the C programming language and include:

- integer
- double-precision floating point
- character
- string
- boolean

## Constants

Constants are one type of *operand* that may be used in NightTrace expressions.

Integer literals may be expressed using typical C language notation:

- decimal literals have no special prefix
- octal literals begin with a zero
- hexadecimal literals begin with a 0x

Floating point literals are always considered to be double-precision floating point literals.

String literals must be enclosed within double quotes; to include a double quote in a constant string literal, precede the double quote with a backslash character. For example:

```
“possible \“meltdown\” alert”
```

The case-insensitive boolean constants TRUE and FALSE have the values 1 and 0, respectively.



Table 11-1 shows units and suffixes for time constants.

**Table 11-1. Time Units and Constant Suffixes**

Time Unit	Suffix
Seconds (This is the default)	s
Milliseconds (10e-3 seconds)	ms
Microseconds (10e-6 seconds)	us
Nanoseconds (10e-9 seconds)	ns

## Functions

Functions are pre-defined NightTrace entities that may be used in an *expression*. NightTrace defines five classes of functions:

- trace event functions (see “Trace Event Functions” on page 11-14)
- state functions (see “State Functions” on page 11-37)
- offset functions (see “Offset Functions” on page 11-77)
- summary functions (see “Summary Functions” on page 11-97)
- format and table functions (see “Format and Table Functions” on page 11-104)

The general syntax of all function calls except summary, format, and table functions is as follows. (Optional parts of function calls are in brackets ([]).)

*function\_name* [ ( [parameter] ) ]

The prefix of the *function\_name* determines its class as follows:

offset_	Functions with this prefix provide information about the trace event at the specified <i>offset</i> (or ordinal trace event number). See “Offset Functions” on page 11-77.
start_	Functions with this prefix provide information about the <i>start event</i> of the <i>most recent instance of a state</i> . See “Start Functions” on page 11-37.
end_	Functions with this prefix provide information about the <i>end event</i> of the <i>last completed instance of a state</i> . See “End Functions” on page 11-55.
state_	Functions with this prefix provide information about instances of states. See “Multi-State Functions” on page 11-73.
event_	Functions with this prefix provide information about instances of events. See “Multi-Event Functions” on page 11-35.

Some functions can be optionally suffixed by a number, *N*, which specifies the *N*th argument logged with the trace event. *N* defaults to 1 and can have the values 1 through the maximum argument logged. For example,

arg()	Returns the first argument
arg1()	Returns the first argument
arg3()	Returns the third argument
start_id()	Returns a trace event ID
state_gap()	Returns the time between instances of a state

Table 11-2 contains a complete list of functions.

**Table 11-2. NightTrace Functions**

Syntax	Return Type
<pre> id [[QE]] start_id [[QS]] end_id [[QS]] offset_id (offset_expr) </pre>	The integer <i>trace event ID</i> .
<pre> arg[N] [[QE]] start_arg[N] [[QS]] end_arg[N] [[QS]] offset_arg[N] (offset_expr) </pre>	The integer <i>trace event argument</i> .
<pre> arg[N]_dbl [[QE]] start_arg[N]_dbl [[QS]] end_arg[N]_dbl [[QS]] offset_arg[N]_dbl (offset_expr) </pre>	The double-precision floating point <i>trace event argument</i> .
<pre> num_args [[QE]] start_num_args [[QS]] end_num_args [[QS]] offset_num_args (offset_expr) </pre>	The number of arguments associated with a <i>trace event</i> .
<pre> pid [[QE]] start_pid [[QS]] end_pid [[QS]] offset_pid (offset_expr) </pre>	The integer global process identifier ( <i>PID</i> ) associated with a <i>trace event</i> .
<pre> raw_pid [[QE]] start_raw_pid [[QS]] end_raw_pid [[QS]] offset_raw_pid (offset_expr) </pre>	The integer process identifier ( <i>raw PID</i> ) associated with a <i>trace event</i> .
<pre> lwpid [[QE]] start_lwpid [[QS]] end_lwpid [[QS]] offset_lwpid (offset_expr) </pre>	The integer lightweight process identifier ( <i>LWPID</i> ) associated with a <i>trace event</i> .
<pre> thread_id [[QE]] start_thread_id [[QS]] end_thread_id [[QS]] offset_thread_id (offset_expr) </pre>	The integer <i>thread identifier (thread ID)</i> associated with a <i>trace event</i> .
<pre> task_id [[QE]] start_task_id [[QS]] end_task_id [[QS]] offset_task_id (offset_expr) </pre>	The integer Ada task identifier associated with a <i>trace event</i> .
<pre> tid [[QE]] start_tid [[QS]] end_tid [[QS]] offset_tid (offset_expr) </pre>	The integer NightTrace thread identifier ( <i>TID</i> ) associated with a <i>trace event</i> .

Table 11-2. NightTrace Functions

Syntax	Return Type
<pre> cpu [[QE]] start_cpu [[QS]] end_cpu [[QS]] offset_cpu (offset_expr) </pre>	The integer logical CPU number associated with a <i>trace event</i> . This function is only valid when applied to events from Night-Trace kernel trace event files.
<pre> time [[QE]] start_time [[QS]] end_time [[QS]] offset_time (offset_expr) </pre>	The double-precision floating point time, expressed in units of seconds, between a <i>trace event</i> and the earliest trace event from all <i>trace event files</i> currently in use.
<pre> node_id [[QE]] start_node_id [[QS]] end_node_id [[QS]] offset_node_id (offset_expr) </pre>	The internally-assigned integer <i>node identifier</i> associated with a <i>trace event</i> .
<pre> pid_table_name [[QE]] start_pid_table_name [[QS]] end_pid_table_name [[QS]] offset_pid_table_name (offset_expr) </pre>	The string describing the name of the process identifier table ( <i>PID table</i> ) associated with a <i>trace event</i> .
<pre> tid_table_name [[QE]] start_tid_table_name [[QS]] end_tid_table_name [[QS]] offset_tid_table_name (offset_expr) </pre>	The string describing the name of the internally-assigned thread identifier table ( <i>TID table</i> ) associated with a <i>trace event</i> .
<pre> node_name [[QE]] start_node_name [[QS]] end_node_name [[QS]] offset_node_name (offset_expr) </pre>	The string describing the name of the system from which a <i>trace event</i> was logged.
<pre> process_name [[QE]] offset_process_name (offset_expr) </pre>	The string describing the name of the process ( <i>PID</i> ) associated with a <i>trace event</i> .
<pre> task_name [[QE]] offset_task_name (offset_expr) </pre>	The string describing the name of the Ada <i>task</i> associated with a <i>trace event</i> .
<pre> thread_name [[QE]] offset_thread_name (offset_expr) </pre>	The string describing the name of the C <i>thread</i> associated with a <i>trace event</i> .
<pre> event_gap [[QE]] state_gap [[QS]] </pre>	The double-precision floating point time, expressed in units of seconds, between the instances of either a <i>trace event</i> or a <i>state</i> .
<pre> state_dur [[QS]] </pre>	The double-precision floating point time, expressed in units of seconds, of an instance of a <i>state</i> .
<pre> event_matches [[QE]] state_matches [[QS]] summary_matches [0] </pre>	The integer number of instances of either a <i>trace event</i> or a <i>state</i> .
<pre> state_status [[QS]] </pre>	The boolean status of a <i>state</i> ; true if the <i>current time line</i> is within an instance of the state, false otherwise. See “state_status()” on page 11-76 for important details.

Table 11-2. NightTrace Functions

Syntax	Return Type
<code>offset</code> <code>[[QE]]</code> <code>start_offset</code> <code>[[QS]]</code> <code>end_offset</code> <code>[[QS]]</code>	The integer ordinal number ( <i>offset</i> ) of a <i>trace event</i> .
<code>min_offset</code> ( <i>expr</i> ) <code>max_offset</code> ( <i>expr</i> )	The integer ordinal number ( <i>offset</i> ) of a <i>trace event</i> associated with a minimum or maximum occurrence of <i>expr</i> .
<code>min</code> ( <i>expr</i> ) <code>max</code> ( <i>expr</i> ) <code>avg</code> ( <i>expr</i> ) <code>sum</code> ( <i>expr</i> )	The minimum, maximum, average, or sum of <i>expr</i> values before the <i>current time</i> . The return type is that of <i>expr</i> .
<code>get_string</code> ( <i>table_name</i> [, <i>int_expr</i> ])	The character string associated with item <i>int_expr</i> in string table <i>table_name</i> .
<code>get_item</code> ( <i>table_name</i> , " <i>str_const</i> ")	The first integer item number associated with string <i>str_const</i> in string table <i>table_name</i> .
<code>get_format</code> ( <i>table_name</i> [, <i>int_expr</i> ])	The character string associated with item <i>int_expr</i> in format table <i>table_name</i> .
<code>format</code> (" <i>format_string</i> " [, <i>arg</i> ] ...)	A character string to format and display.

## Function Parameters

If the function has a *parameter*, the parentheses are required. Otherwise, they are optional. For example,

<code>arg2</code>	No parentheses are required
<code>arg2 ()</code>	No parentheses are required
<code>arg2 (GAK)</code>	Parentheses are required

In many functions, the *parameter* is optional because it can be inferred from context. For trace event functions, the *current trace event* is used if the parameter is omitted. For state functions, the state being defined is used if the parameter is omitted. (Thus, state functions without parameters can only be used inside state definitions). For example,

<code>arg1 ()</code>	Operates on the <i>current trace event</i>
<code>arg1 (my_event)</code>	Operates on the <i>qualified event</i> <i>my_event</i>
<code>end_arg1 ()</code>	Operates on the <i>last completed instance</i> of the state being defined and can only appear within a state definition

`end_arg1(my_state)` Operates on the *last completed instance* of the *qualified state* `my_state`

This manual uses the following conventions for function *parameters*:

<i>QE</i>	A user-defined <i>qualified event</i> . If supplied, the function applies to the specified qualified event. For more information, see “Qualified Events” on page 11-113.
<i>QS</i>	A user-defined <i>qualified state</i> . If supplied, the function applies to the specified qualified state. For more information, see “Qualified States” on page 11-116.
<i>offset_expr</i>	An expression that evaluates to the <i>offset</i> (or ordinal trace event number) of a trace event.
<i>expr</i>	Any valid NightTrace <i>expression</i> (see “Expressions” on page 11-1).
<i>table_name</i>	An unquoted character string that represents the name of a <i>string table</i> or <i>format table</i> .
<i>int_expr</i>	An integer expression that acts as an index into the specified <i>string table</i> or <i>format table</i> . <i>int_expr</i> must either match an identifying integer value in the <i>table_name</i> table, or the <i>table_name</i> table must have a <code>default item</code> line.
<i>str_const</i>	A string constant literal that acts as an index into the specified <i>string table</i> .
<i>format_string</i>	A character string that contains literal characters and conversion specifications. Conversion specifications modify zero or more <i>args</i> .
<i>arg</i>	An optional expression to be formatted and displayed.

#### NOTE

NightTrace does not perform semantic error checking of functions. For example, if you ask for information about the second argument, but no second argument was logged, NightTrace does not tell you. Similarly, NightTrace does not flag the use of undefined *macros*, *qualified events*, and *qualified states*.

## Function Terminology

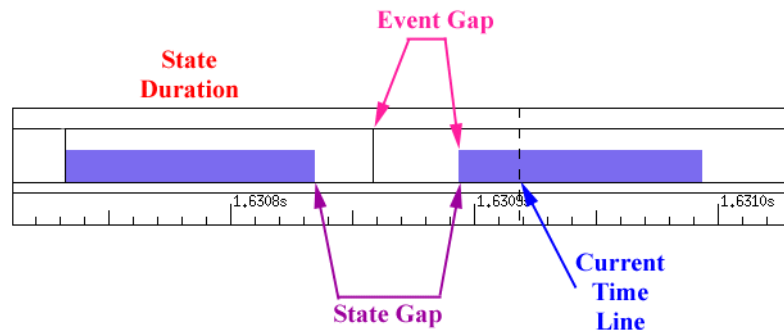
In order to use the NightTrace functions effectively, it may be useful to understand some of the concepts associated with them.

A *trace event* represents a user-defined or kernel-defined event, logged with optional data arguments. Events are given discrete numbers to identify them; this number is called the *trace event ID*. A *state* is defined to be the interval of time between two specific events.

The descriptions of the functions further speak in terms of “instances” of states. These are best defined as:

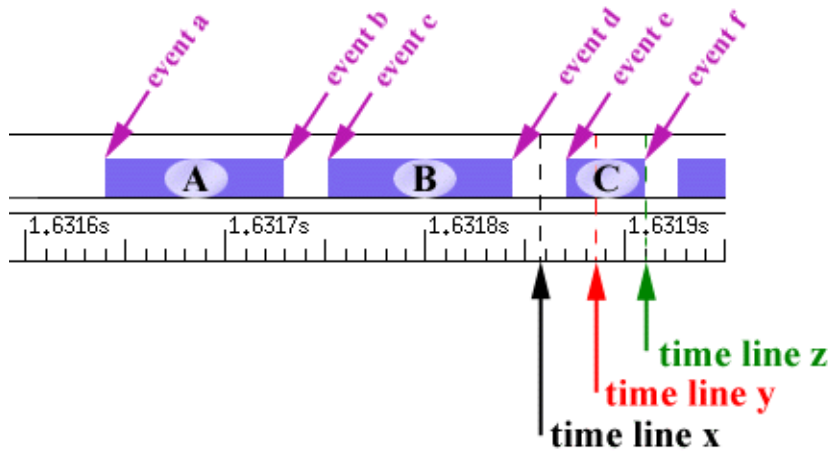
<i>current instance</i>	The instance of a state which has begun but has not yet completed. Thus, the <i>current time line</i> would be positioned within the region from the <i>start event</i> up to, but not including, the <i>end event</i> .
<i>last completed instance</i>	The most recent instance of a state that has already completed. Thus, the <i>current time line</i> would be positioned either on, or after, the <i>end event</i> for a state.
<i>most recent instance</i>	If the <i>current time line</i> is positioned within a current instance of a state, then it is that instance of the state. Otherwise, it is the last completed instance of a state.

Figure 11-1 illustrates some of these concepts with a State Graph.



**Figure 11-1. Function Terminology Illustrated**

A more detailed example is illustrated in Figure 11-2.



**Figure 11-2. States and Events**

The following discusses the terminology with respect to **time line x**, **time line y**, and **time line z**.

Assuming the current time line was positioned at **time line x** in Figure 11-2, the various “instances” would be defined as:

<i>current instance</i>	No current instance is defined since the current time line is not positioned within any instance of a state.
<i>last completed instance</i>	Instance B
<i>most recent instance</i>	Instance B. Since the current time line is not positioned within any instance of a state, the most recent instance is the last completed instance.



The table below indicates the information returned by various NightTrace functions assuming the current time line was positioned at **time line x** in Figure 11-2.

<code>state_status()</code>	false	The current time line was not positioned within a current instance of a state.
<code>state_gap()</code>	~0.000020	The duration of time in seconds between event b and event c. The function operated the most recent instance of the state (instance B) and the immediately preceding instance (instance A).
<code>state_dur()</code>	~0.000090	The duration of time in seconds between event c and event d. The function operated on the last completed instance of the state (instance B).
<code>state_matches()</code>	2	Assuming no other instances of the state preceded those shown in the figure. The function operated on all completed instances of the state (which included instances A and B).
<code>start_time()</code>	~1.631750	The time associated with event c. The function operated on the most recent instance of the state (instance B).
<code>end_time()</code>	~1.631840	The time associated with event d. The function operated on the last completed instance of the state (instance B).

Assuming the current time line was positioned at **time line y** in Figure 11-2, the various “instances” would be defined as:

<i>current instance</i>	Instance C
<i>last completed instance</i>	Instance B
<i>most recent instance</i>	Instance C

The table below indicates the information returned by various NightTrace functions assuming the current time line was positioned at **time line y** in Figure 11-2.

<code>state_status()</code>	true	The current time line was positioned inside a current instance of the state (instance C).
<code>state_gap()</code>	~0.000030	The duration of time in seconds between event d and event e. The function operated on the most recent instance of the state (instance C) and the immediately preceding instance (instance B).
<code>state_dur()</code>	~0.000090	The duration of time in seconds between event c and event d. The function operated on the last completed instance of the state (instance B).
<code>state_matches()</code>	2	Assuming no other instances of the state preceded those shown in the figure. The function operated on all completed instances of the state (which included instances A and B).
<code>start_time()</code>	~1.631870	The time associated with event e. The function operated on the most recent instance of the state (instance C).
<code>end_time()</code>	~1.631840	The time associated with event d. The function operated on the last completed instance of the state (instance B).

Assuming the current time line was positioned at **time line z** in Figure 11-2, the various “instances” would be defined as:

<i>current instance</i>	No current instance is defined since the current time line is positioned on the <i>end event</i> of an instance of a state.
<i>last completed instance</i>	Instance C
<i>most recent instance</i>	Instance C

The table below indicates the information returned by various NightTrace functions assuming the current time line was positioned at **time line z** in Figure 11-2.

<code>state_status()</code>	false	The current time line was not positioned inside a current instance of the state. Even though the current time line is positioned on an <i>end event</i> of the state (event f), the corresponding instance is said to have already completed.
<code>state_gap()</code>	~0.000030	The duration of time in seconds between event d and event e. The function operated on the most recent instance of the state (instance C) and the immediately preceding instance (instance B).
<code>state_dur()</code>	~0.000040	The duration of time in seconds between event e and event f. The function operated on the last completed instance of the state (instance C).
<code>state_matches()</code>	3	Assuming no other instances of the state preceded those shown in the figure. The function operated on all completed instances of the state (which included instances A, B, and C).
<code>start_time()</code>	~1.631870	The time associated with event e. The function operated on the most recent instance of the state (instance C).
<code>end_time()</code>	~1.631910	The time associated with event f. The function operated on the last completed instance of the state (instance C).

## Trace Event Functions

The trace event functions operate on either the *qualified event* specified to that function or the *current trace event*. They include the following:

- `id()`
- `arg()`
- `arg_dbl()`
- `num_args()`
- `pid()`
- `raw_pid()`
- `lwpid()`
- `cpu()`
- `thread_id()`
- `task_id()`
- `tid()`
- `offset()`
- `time()`
- `node_id()`
- `pid_table_name()`
- `tid_table_name()`
- `node_name()`
- `process_name()`
- `task_name()`
- `thread_name()`
- **Multi-event functions**

**id()****DESCRIPTION**

The `id()` function returns the *trace event ID* of the last instance of a *trace event*.

**SYNTAX**

```
id [(QE)]
```

**PARAMETERS**

*QE*

A user-defined *qualified event*. If supplied, the function returns the *trace event ID* of the last instance of the trace event which satisfies the conditions of the specified qualified event. If omitted, the function returns the *trace event ID* of the current trace event. For more information, see “Qualified Events” on page 11-113.

**RETURN TYPE**

integer

**SEE ALSO**

- “start\_id()” on page 11-38
- “end\_id()” on page 11-56
- “offset\_id()” on page 11-78

## arg()

### DESCRIPTION

The `arg()` function returns the value of a particular *trace event argument*.

### SYNTAX

`arg[N] [(QE)]`

### PARAMETERS

<i>N</i>	Specifies the <i>N</i> th argument logged with the <i>trace event</i> . Defaults to 1.
<i>QE</i>	A user-defined <i>qualified event</i> . If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the specified argument for the <i>current trace event</i> . For more information, see “Qualified Events” on page 11-113.

### RETURN TYPE

integer

### SEE ALSO

- “arg\_dbl()” on page 11-17
- “num\_args()” on page 11-18
- “start\_arg()” on page 11-39
- “end\_arg()” on page 11-57
- “offset\_arg()” on page 11-79

**arg\_dbl()****DESCRIPTION**

The `arg_dbl()` function returns the value of a particular *trace event argument*.

**SYNTAX**

```
arg[N]_dbl [(QE)]
```

**PARAMETERS**

<i>N</i>	Specifies the <i>N</i> th argument logged with the <i>trace event</i> . Defaults to 1.
<i>QE</i>	A user-defined <i>qualified event</i> . If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the specified argument for the <i>current trace event</i> . For more information, see “Qualified Events” on page 11-113.

**RETURN TYPE**

double-precision floating point

**SEE ALSO**

- “arg()” on page 11-16
- “num\_args()” on page 11-18
- “start\_arg\_dbl()” on page 11-40
- “end\_arg\_dbl()” on page 11-58
- “offset\_arg\_dbl()” on page 11-80

## num\_args()

### DESCRIPTION

The `num_args()` function returns the number of arguments logged with a *trace event*.

### SYNTAX

```
num_args [(QE)]
```

### PARAMETERS

*QE* A user-defined *qualified event*. If supplied, the function returns the number of arguments of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the number of arguments of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

### RETURN TYPE

integer

### SEE ALSO

- “arg()” on page 11-16
- “start\_num\_args()” on page 11-41
- “end\_num\_args()” on page 11-59
- “offset\_num\_args()” on page 11-81



**pid()****DESCRIPTION**

The `pid()` function returns the global process identifier (*PID*) associated with a *trace event*.

**NOTE**

On PowerMAX OS systems, a global process identifier does not have the same meaning as the typical operating system definition of `pid`. A PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. Consult the `_lwp_global_self(2)` man page for more information.

On Linux systems, the `pid()` is the same as the operating system process identifier.

**SYNTAX**

```
pid [[QE]]
```

**PARAMETERS**

*QE*                      A user-defined *qualified event*. If supplied, the function returns the global process identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the global process identifier of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

**RETURN TYPE**

integer

**SEE ALSO**

- “Processes” on page 10-51
- “raw\_pid()” on page 11-20
- “lwpid()” on page 11-21
- “start\_pid()” on page 11-42
- “end\_pid()” on page 11-60
- “offset\_pid()” on page 11-82

## raw\_pid()

### DESCRIPTION

The `raw_pid()` function returns the process identifier (*raw PID*) associated with a *trace event*.

### NOTE

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. The `raw_pid()` function returns the upper 16 bits.

On Linux systems, the `raw_pid()` is the same as the operating system process identifier.

### SYNTAX

```
raw_pid([(QE)])
```

### PARAMETERS

*QE* A user-defined *qualified event*. If supplied, the function returns the process identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the process identifier of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

### RETURN TYPE

integer

### SEE ALSO

- “Processes” on page 10-51
- “pid()” on page 11-19
- “lwpid()” on page 11-21
- “start\_raw\_pid()” on page 11-43
- “end\_raw\_pid()” on page 11-61
- “offset\_raw\_pid()” on page 11-83

**lwpid()****DESCRIPTION**

The `lwpid()` function returns the lightweight process identifier (*LWPID*) associated with a *trace event*.

**NOTE**

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. The `lwpid()` function returns the lower 16 bits. See the `_lwp_self(2)` man page for more information.

On Linux systems, the `lwpid()` is the same as the operating system process identifier.

**SYNTAX**

```
lwpid([(QE)])
```

**PARAMETERS**

*QE*                      A user-defined *qualified event*. If supplied, the function returns the lightweight process identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the lightweight process identifier of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

**RETURN TYPE**

integer

**SEE ALSO**

- “Processes” on page 10-51
- “pid()” on page 11-19
- “raw\_pid()” on page 11-20
- “start\_lwpid()” on page 11-44
- “end\_lwpid()” on page 11-62
- “offset\_lwpid()” on page 11-84

## thread\_id()

### DESCRIPTION

The `thread_id()` function returns the *thread* identifier associated with a *trace event*.

### NOTE

See the `thr_self(3thread)` man page for more information.

### SYNTAX

```
thread_id([(QE)])
```

### PARAMETERS

*QE* A user-defined *qualified event*. If supplied, the function returns the thread identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the thread identifier of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

### RETURN TYPE

integer

### SEE ALSO

- “start\_thread\_id()” on page 11-45
- “end\_thread\_id()” on page 11-63
- “offset\_thread\_id()” on page 11-85

**task\_id()****DESCRIPTION**

The `task_id()` function returns the Ada task identifier associated with a *trace event*.

**NOTE**

This function is only meaningful for trace events logged by Ada tasking programs.

**SYNTAX**

```
task_id [[QE]]
```

**PARAMETERS**

*QE*                    A user-defined *qualified event*. If supplied, the function returns the Ada task identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the Ada task identifier of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

**RETURN TYPE**

integer

**SEE ALSO**

- “start\_task\_id()” on page 11-46
- “end\_task\_id()” on page 11-64
- “offset\_task\_id()” on page 11-86

## tid()

### DESCRIPTION

The `tid()` function returns the internally-assigned NightTrace thread identifier (*TID*) associated with a *trace event*.

### SYNTAX

```
tid [[QE]]
```

### PARAMETERS

*QE* A user-defined *qualified event*. If supplied, the function returns the NightTrace thread identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the NightTrace thread identifier of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

### RETURN TYPE

integer

### SEE ALSO

- “Threads” on page 10-52
- “start\_tid()” on page 11-47
- “end\_tid()” on page 11-65
- “offset\_tid()” on page 11-87

**cpu()****DESCRIPTION**

The `cpu ()` function returns the logical CPU number associated with a *trace event*. CPUs are logically numbered starting at 0 and monotonically increase thereafter.

**NOTE**

This function is only valid when applied to events from Night-Trace kernel trace event files.

**SYNTAX**

`cpu [[QE]]`

**PARAMETERS**

*QE* A user-defined *qualified event*. If supplied, the function returns the logical CPU number of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the logical CPU number of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

**RETURN TYPE**

integer

**SEE ALSO**

- “start\_cpu()” on page 11-48
- “end\_cpu()” on page 11-66
- “offset\_cpu()” on page 11-88

## offset()

### DESCRIPTION

The `offset()` function returns the ordinal number (*offset*) of a *trace event*.

### SYNTAX

```
offset [(QE)]
```

### PARAMETERS

*QE* A user-defined *qualified event*. If supplied, the function returns the ordinal number (*offset*) of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the ordinal number (*offset*) of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

### RETURN TYPE

integer

### SEE ALSO

- “start\_offset()” on page 11-49
- “end\_offset()” on page 11-67
- “min\_offset()” on page 11-101
- “max\_offset()” on page 11-102



**time()****DESCRIPTION**

The `time()` function returns the time, in seconds, associated with a *trace event*. Times are relative to the earliest trace event from all trace data files currently in use.

**SYNTAX**

```
time [[QE]]
```

**PARAMETERS**

*QE* A user-defined *qualified event*. If supplied, the function returns the time, in seconds, of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the time, in seconds, of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

**RETURN TYPE**

double-precision floating point

**SEE ALSO**

- “`event_gap()`” on page 11-35
- “`start_time()`” on page 11-50
- “`end_time()`” on page 11-68
- “`state_gap()`” on page 11-73
- “`state_dur()`” on page 11-74
- “`offset_time()`” on page 11-89

## node\_id()

### DESCRIPTION

The `node_id()` function returns the internally-assigned *node identifier* associated with a *trace event*.

### NOTE

The `node_id()` function is of limited usefulness since the node identifier is an internally-assigned integer number assigned by NightTrace. The `node_name()` function is more useful, as it returns the name of the system from which a trace event was logged. (See “`node_name()`” on page 11-31 for more information about this function.)

### SYNTAX

```
node_id([(QE)])
```

### PARAMETERS

*QE*                      A user-defined *qualified event*. If supplied, the function returns the node identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the node identifier of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

### RETURN TYPE

integer

### SEE ALSO

- “`start_node_id()`” on page 11-51
- “`offset_node_id()`” on page 11-90
- “`end_node_id()`” on page 11-69

**pid\_table\_name()****DESCRIPTION**

The `pid_table_name()` function returns the name of the internally-assigned NightTrace process identifier table (*PID table*) associated with a *trace event*.

**SYNTAX**

```
pid_table_name [(QE)]
```

**PARAMETERS**

*QE* A user-defined *qualified event*. If supplied, the function returns the name of the process identifier table (*PID table*) of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the name of the process identifier table (*PID table*) of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

**RETURN TYPE**

string

**SEE ALSO**

- “start\_pid\_table\_name()” on page 11-52
- “offset\_pid\_table\_name()” on page 11-91
- “end\_pid\_table\_name()” on page 11-70

## tid\_table\_name()

### DESCRIPTION

The `tid_table_name()` function returns the name of the internally-assigned NightTrace thread identifier table (*TID table*) associated with a *trace event*.

### SYNTAX

```
tid_table_name [(QE)]
```

### PARAMETERS

*QE* A user-defined *qualified event*. If supplied, the function returns the name of the thread identifier table (*TID table*) of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the name of the thread identifier table (*TID table*) of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

### RETURN TYPE

string

### SEE ALSO

- “start\_tid\_table\_name()” on page 11-53
- “offset\_tid\_table\_name()” on page 11-92
- “end\_tid\_table\_name()” on page 11-71

**node\_name()****DESCRIPTION**

The `node_name ()` function returns the name of the system from which a *trace event* was logged.

**SYNTAX**

`node_name [(QE)]`

**PARAMETERS**

*QE* A user-defined *qualified event*. If supplied, the function returns the name of system from which the last instance of the trace event which satisfies the conditions for the specified qualified event was logged. If omitted, the function returns the name of the system from which the *current trace event* was logged. For more information, see “Qualified Events” on page 11-113.

**RETURN TYPE**

string

**SEE ALSO**

- “start\_node\_name()” on page 11-54
- “offset\_node\_name()” on page 11-93
- “end\_node\_name()” on page 11-72

## process\_name()

### DESCRIPTION

The `process_name()` function returns the name of the process (*PID*) associated with a *trace event*.

See “Processes” on page 11-127 for a discussion of the usage of this function.

### SYNTAX

```
process_name [[QE]]
```

### PARAMETERS

*QE*                      A user-defined *qualified event*. If supplied, the function returns the name associated with the *PID* of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the name associated with the *PID* of the *current trace event*. For more information, see “Qualified Events” on page 11-113.

### RETURN TYPE

string

### SEE ALSO

- “`offset_process_name()`” on page 11-94

**task\_name()****DESCRIPTION**

The `task_name()` function returns the name of the task associated with a *trace event*.

**NOTE**

This function is only meaningful for trace events which were logged from Ada tasking programs.

**SYNTAX**

`task_name` `[[QE]]`

**PARAMETERS**

*QE*                      A user-defined *qualified event*. If supplied, the function returns the name of the task associated with the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the name of the task associated with the *current trace event*. For more information, see “Qualified Events” on page 11-113.

**RETURN TYPE**

string

**SEE ALSO**

- “`offset_task_name()`” on page 11-95

## thread\_name()

### DESCRIPTION

The `thread_name()` function returns the thread name associated with a *trace event*.

See “Threads” on page 11-128 for a discussion of the usage of this function.

### SYNTAX

```
thread_name [[QE]]
```

### PARAMETERS

*QE*                      A user-defined *qualified event*. If supplied, the function returns the thread name associated with the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the thread name associated with the *current trace event*. For more information, see “Qualified Events” on page 11-113.

### RETURN TYPE

string

### SEE ALSO

- “`offset_thread_name()`” on page 11-96



## Multi-Event Functions

Multi-event functions return information about one or more instances of an event:

- `event_gap()`
- `event_matches()`

`event_gap()`

### DESCRIPTION

The `event_gap()` function returns the time, in seconds, between the most recent occurrence of a specific event and its immediately preceding occurrence.

### SYNTAX

```
event_gap [(QE)]
```

### PARAMETERS

<i>QE</i>	A user-defined <i>qualified event</i> . If supplied, the function calculates the gap between the two most recent occurrences of events which satisfy the conditions of the specified qualified event. If omitted, the function calculates the gap between the current trace event and the event immediately preceding it. For more information, see “Qualified Events” on page 11-113.
-----------	--

### RETURN TYPE

double-precision floating point

### SEE ALSO

- “`time()`” on page 11-27
- “`state_gap()`” on page 11-73
- “`state_dur()`” on page 11-74

event\_matches()

### DESCRIPTION

The `event_matches()` function returns the number of occurrences of a *trace event* on or before the *current time line*.

### SYNTAX

```
event_matches [[QE]]
```

### PARAMETERS

*QE*

A user-defined *qualified event*. If supplied, the function calculates the number of occurrences of events which satisfy the conditions of the specified qualified event on or before the current time line. If omitted, the function calculates the number of occurrences of all events on or before the current time line. For more information, see “Qualified Events” on page 11-113.

### RETURN TYPE

integer

### SEE ALSO

- “summary\_matches()” on page 11-103

## State Functions

In its simplest form, a *state* is a region of time bounded by two *trace events*. A state definition requires the specification of two trace events, a *start event* and an *end event*, respectively. Additional conditions may be specified in a state definition to further constrain the state. The state functions include the following:

- start functions (see “Start Functions” on page 11-37)
- end functions (see “End Functions” on page 11-55)
- multi-state functions (see “Multi-State Functions” on page 11-73)

### NOTE

Currently, NightTrace does not supported nesting of states. Thus, once the conditions which satisfy a *start event* are met, no other instances of that state can begin until the end condition has been met.

## Start Functions

The start functions provide information about the *start event* of the *most recent instance of a state*. The state to which the start function applies is either the *qualified state* specified to the function, or the state being currently defined. Thus, if a qualified state is not specified, start functions are only meaningful when used in *expressions* associated within a state definition. In addition, start functions should not be used in a recursive manner in a **Start Expression**; a start function should not be specified in a **Start Expression** that applies to the state definition containing that **Start Expression**. Conversely, an **End Expression** may include start functions that apply to the state definition containing that **End Expression**.

### NOTE

Start functions provide information about the *most recent instance of a state*, whereas end functions (see “End Functions” on page 11-55) provide information about the *last completed instance of a state*.

Start functions include the following:

- `start_id()`
- `start_arg()`
- `start_arg_dbl()`
- `start_num_args()`
- `start_pid()`
- `start_raw_pid()`

- `start_thread_id()`
- `start_task_id()`
- `start_tid()`
- `start_lwpid()`
- `start_cpu()`
- `start_offset()`
- `start_time()`
- `start_node_id()`
- `start_pid_table_name()`
- `start_tid_table_name()`
- `start_node_name()`

`start_id()`

## DESCRIPTION

The `start_id()` function returns the *trace event ID* of the *start event* of the *most recent instance of a state*.

## SYNTAX

```
start_id [[QS]]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “`id()`” on page 11-15
- “`end_id()`” on page 11-56
- “`offset_id()`” on page 11-78

start\_arg()

## DESCRIPTION

The `start_arg()` function returns the value of a particular *trace event argument* associated with the *start event* of the *most recent instance of a state*.

## SYNTAX

```
start_arg[N] [(QS)]
```

## PARAMETERS

<i>N</i>	Specifies the <i>N</i> th argument logged with the <i>start event</i> . Defaults to 1.
<i>QS</i>	A user-defined <i>qualified state</i> . If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “arg()” on page 11-16
- “start\_arg\_dbl()” on page 11-40
- “start\_num\_args()” on page 11-41
- “end\_arg()” on page 11-57
- “offset\_arg()” on page 11-79

start\_arg\_dbl()

## DESCRIPTION

The `start_arg_dbl()` function returns the value of a particular *trace event argument* associated with the *start event* of the *most recent instance of a state*.

## SYNTAX

```
start_arg[N]_dbl [(QS)]
```

## PARAMETERS

<i>N</i>	Specifies the <i>N</i> th argument logged with the <i>start event</i> . Defaults to 1.
<i>QS</i>	A user-defined <i>qualified state</i> . If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

double-precision floating point

## SEE ALSO

- “arg\_dbl()” on page 11-17
- “start\_arg()” on page 11-39
- “start\_num\_args()” on page 11-41
- “end\_arg\_dbl()” on page 11-58
- “offset\_arg\_dbl()” on page 11-80

start\_num\_args()

## DESCRIPTION

The `start_num_args()` function returns the number of arguments associated with the *start event* of the *most recent instance of a state*.

## SYNTAX

```
start_num_args [(QS)]
```

## PARAMETERS

<i>QS</i>	A user-defined <i>qualified state</i> . If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.
-----------	---

## RETURN TYPE

integer

## SEE ALSO

- “start\_arg()” on page 11-39
- “num\_args()” on page 11-18
- “end\_num\_args()” on page 11-59
- “offset\_num\_args()” on page 11-81

start\_pid()

## DESCRIPTION

The `start_pid()` function returns the PID associated with the *start event* of the *most recent instance of a state*.

## NOTE

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. Consult the `_lwp_global_self(2)` man page for more information.

On Linux systems, the PID is the same as the operating system process identifier.

## SYNTAX

```
start_pid([(QS)])
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “pid()” on page 11-19
- “raw\_pid()” on page 11-20
- “lwpid()” on page 11-21
- “end\_pid()” on page 11-60
- “offset\_pid()” on page 11-82



start\_raw\_pid()

## DESCRIPTION

The `start_raw_pid()` function returns the process identifier (*raw PID*) associated with the *start event* of the *most recent instance of a state*.

## NOTE

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. The `start_raw_pid()` function returns the upper 16 bits.

On Linux systems, the `start_raw_pid()` returns the operating system process identifier.

## SYNTAX

```
start_raw_pid [[QS]]
```

## PARAMETERS

*QS*                      A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “pid()” on page 11-19
- “raw\_pid()” on page 11-20
- “lwpid()” on page 11-21
- “end\_pid()” on page 11-60
- “offset\_pid()” on page 11-82

start\_lwpid()

## DESCRIPTION

The `start_lwpid()` function returns the lightweight process identifier (*LWPID*) associated with the *start event* of the *most recent instance of a state*.

## NOTE

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. The `start_lwpid()` function returns the lower 16 bits. See the `_lwp_self(2)` man page for more information.

On Linux systems, `start_lwpid()` returns the operating system process identifier.

## SYNTAX

```
start_lwpid [[QS]]
```

## PARAMETERS

*QS*                    A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “pid()” on page 11-19
- “raw\_pid()” on page 11-20
- “lwpid()” on page 11-21
- “end\_pid()” on page 11-60
- “offset\_pid()” on page 11-82

`start_thread_id()`**DESCRIPTION**

The `start_thread_id()` function returns the *thread* identifier associated with the *start event* of the *most recent instance of a state*.

**NOTE**

See the `thr_self(3thread)` man page for more information.

**SYNTAX**

```
start_thread_id [[QS]]
```

**PARAMETERS**

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

**RETURN TYPE**

integer

**SEE ALSO**

- “`thread_id()`” on page 11-22
- “`end_thread_id()`” on page 11-63
- “`offset_thread_id()`” on page 11-85

start\_task\_id()

## DESCRIPTION

The `start_task_id()` function returns the Ada task identifier associated with the *start event* of the *most recent instance of a state*.

## NOTE

This function is only meaningful for trace events logged by Ada tasking programs.

## SYNTAX

```
start_task_id [[QS]]
```

## PARAMETERS

*QS*                      A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “task\_id()” on page 11-23
- “end\_task\_id()” on page 11-64
- “offset\_task\_id()” on page 11-86

`start_tid()`**DESCRIPTION**

The `start_tid()` function returns the internally-assigned NightTrace thread identifier (*TID*) associated with the *start event* of the *most recent instance of a state*.

**SYNTAX**

```
start_tid([(QS)])
```

**PARAMETERS**

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

**RETURN TYPE**

integer

**SEE ALSO**

- “tid()” on page 11-24
- “end\_tid()” on page 11-65
- “offset\_tid()” on page 11-87

start\_cpu()

## DESCRIPTION

The `start_cpu()` function returns the logical CPU number associated with the *start event* of the *most recent instance of a state*. CPUs are logically numbered starting at 0 and monotonically increase thereafter.

## NOTE

This function is only valid when applied to events from Night-Trace kernel trace event files.

## SYNTAX

```
start_cpu [[QS]]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “cpu()” on page 11-25
- “end\_cpu()” on page 11-66
- “offset\_cpu()” on page 11-88

`start_offset()`**DESCRIPTION**

The `start_offset()` function returns the ordinal number (*offset*) of the *start event* of the *most recent instance of a state*.

**SYNTAX**

```
start_offset [(QS)]
```

**PARAMETERS**

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

**RETURN TYPE**

integer

**SEE ALSO**

- “offset()” on page 11-26
- “end\_offset()” on page 11-67

start\_time()

## DESCRIPTION

The `start_time()` function returns the time, in seconds, associated with the *start event* of the *most recent instance of a state*. Times are relative to the earliest trace event from all trace data files currently in use.

## SYNTAX

```
start_time [(QS)]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

double-precision floating point

## SEE ALSO

- “time()” on page 11-27
- “end\_time()” on page 11-68
- “state\_gap()” on page 11-73
- “state\_dur()” on page 11-74
- “offset\_time()” on page 11-89



`start_node_id()`**DESCRIPTION**

The `start_node_id()` function returns the internally-assigned *node identifier* associated with the *start event* of the *most recent instance of a state*.

**SYNTAX**

```
start_node_id [[QS]]
```

**PARAMETERS**

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

**RETURN TYPE**

integer

**SEE ALSO**

- “`node_id()`” on page 11-28
- “`offset_node_id()`” on page 11-90
- “`end_node_id()`” on page 11-69

start\_pid\_table\_name()

### DESCRIPTION

The `start_pid_table_name()` function returns the name of the internally-assigned NightTrace process identifier table (*PID table*) associated with the *start event* of the *most recent instance of a state*.

### SYNTAX

```
start_pid_table_name [[QS]]
```

### PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

### RETURN TYPE

string

### SEE ALSO

- “pid\_table\_name()” on page 11-29
- “offset\_pid\_table\_name()” on page 11-91
- “end\_pid\_table\_name()” on page 11-70

`start_tid_table_name()`

## DESCRIPTION

The `start_tid_table_name()` function returns the name of the internally-assigned NightTrace thread identifier table (*TID table*) associated with the *start event* of the *most recent instance of a state*.

## SYNTAX

```
start_tid_table_name [(QS)]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

string

## SEE ALSO

- “`tid_table_name()`” on page 11-30
- “`offset_tid_table_name()`” on page 11-92
- “`end_tid_table_name()`” on page 11-71

start\_node\_name()

## DESCRIPTION

The `start_node_name()` function returns the name of the system from which the *start event* of the *most recent instance of a state* was logged.

## SYNTAX

```
start_node_name [(QS)]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

string

## SEE ALSO

- “node\_name()” on page 11-31
- “offset\_node\_name()” on page 11-93
- “end\_node\_name()” on page 11-72

## End Functions

The end functions provide information about the *end event* of the *last completed instance of a state*. The *state* to which the end function applies is either the *qualified state* specified to the function, or the state being currently defined. Thus, if a qualified state is not specified, end functions are only meaningful when used in expressions associated within a state definition.

### NOTE

End functions provide information about the *last completed instance of a state*, whereas start functions (see “Start Functions” on page 11-37) provide information about the *most recent instance of a state*.

End functions include:

- `end_id()`
- `end_arg()`
- `end_arg_dbl()`
- `end_num_args()`
- `end_pid()`
- `end_raw_pid()`
- `end_lwpid()`
- `end_thread_id()`
- `end_task_id()`
- `end_tid()`
- `end_cpu()`
- `end_offset()`
- `end_time()`
- `end_node_id()`
- `end_pid_table_name()`
- `end_tid_table_name()`
- `end_node_name()`

end\_id()

## DESCRIPTION

The `end_id()` function returns the *trace event ID* associated with the *end event* of the *last completed instance of a state*.

## SYNTAX

```
end_id [(QS)]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “id()” on page 11-15
- “start\_id()” on page 11-38
- “offset\_id()” on page 11-78

`end_arg()`**DESCRIPTION**

The `end_arg()` function returns the value of a particular *trace event argument* associated with the *end event* of the *last completed instance of a state*.

**SYNTAX**`end_arg[N] [(QS)]`**PARAMETERS**

<i>N</i>	Specifies the <i>N</i> th argument logged with the trace event. Defaults to 1.
<i>QS</i>	A user-defined <i>qualified state</i> . If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

**RETURN TYPE**

integer

**SEE ALSO**

- “arg()” on page 11-16
- “start\_arg()” on page 11-39
- “end\_arg\_dbl()” on page 11-58
- “end\_num\_args()” on page 11-59
- “offset\_arg()” on page 11-79

end\_arg\_dbl()

## DESCRIPTION

The `end_arg_dbl()` function returns the value of a particular *trace event argument* associated with the *end event* of the *last completed instance of a state*.

## SYNTAX

```
end_arg[N]_dbl [(QS)]
```

## PARAMETERS

<i>N</i>	Specifies the <i>N</i> th argument logged with the trace event. Defaults to 1.
<i>QS</i>	A user-defined <i>qualified state</i> . If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

double-precision floating point

## SEE ALSO

- “arg\_dbl()” on page 11-17
- “start\_arg\_dbl()” on page 11-40
- “end\_arg()” on page 11-57
- “end\_num\_args()” on page 11-59
- “offset\_arg\_dbl()” on page 11-80



end\_num\_args()

## DESCRIPTION

The `end_num_args()` function returns the number of arguments associated with the *end event* of the *last completed instance of a state*.

## SYNTAX

```
end_num_args [(QS)]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “num\_args()” on page 11-18
- “start\_num\_args()” on page 11-41
- “end\_arg()” on page 11-57
- “offset\_num\_args()” on page 11-81

end\_pid()

## DESCRIPTION

The `end_pid()` function returns the PID associated with the *end event* of the *last completed instance of a state*.

## NOTE

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. Consult the `_lwp_global_self(2)` man page for more information.

On Linux systems, the PID is the same as the operating system process identifier.

## SYNTAX

```
end_pid [[QS]]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “pid()” on page 11-19
- “raw\_pid()” on page 11-20
- “lwpid()” on page 11-21
- “start\_pid()” on page 11-42
- “offset\_pid()” on page 11-82

end\_raw\_pid()

## DESCRIPTION

The `end_raw_pid()` function returns the process identifier (*raw PID*) associated with the *end event* of the *last completed instance of a state*.

## NOTE

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. The `end_raw_pid()` function returns the upper 16 bits.

On Linux systems, the `end_raw_pid()` is the same as the operating system process identifier.

## SYNTAX

```
end_raw_pid([(QS)])
```

## PARAMETERS

*QS*                      A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “pid()” on page 11-19
- “raw\_pid()” on page 11-20
- “lwpid()” on page 11-21
- “start\_pid()” on page 11-42
- “offset\_pid()” on page 11-82

end\_lwpid()

## DESCRIPTION

The `end_lwpid()` function returns the lightweight process identifier (*LWPID*) associated with the *end event* of the *last completed instance of a state*.

## NOTE

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. The `end_lwpid()` function returns the lower 16 bits. See the `_lwp_self(2)` man page for more information.

On Linux systems, the `end_lwpid()` returns the operating system process identifier.

## SYNTAX

```
end_lwpid([(QS)])
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “pid()” on page 11-19
- “raw\_pid()” on page 11-20
- “lwpid()” on page 11-21
- “start\_pid()” on page 11-42
- “offset\_pid()” on page 11-82

end\_thread\_id()

## DESCRIPTION

The `end_thread_id()` function returns the *thread* identifier associated with the *end event* of the *last completed instance of a state*.

## NOTE

See the `thr_self(3thread)` man page for more information.

## SYNTAX

```
end_thread_id [[QS]]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “thread\_id()” on page 11-22
- “start\_thread\_id()” on page 11-45
- “offset\_thread\_id()” on page 11-85

end\_task\_id()

## DESCRIPTION

The `end_task_id()` function returns the Ada task identifier associated with the *end event* of the *last completed instance of a state*.

## NOTE

This function is only meaningful for trace events logged by Ada tasking programs.

## SYNTAX

`end_task_id` [(*QS*)]

## PARAMETERS

*QS*                      A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “task\_id()” on page 11-23
- “start\_task\_id()” on page 11-46
- “offset\_task\_id()” on page 11-86

`end_tid()`**DESCRIPTION**

The `end_tid()` function returns the internally-assigned NightTrace thread identifier (*TID*) associated with the *end event* of the *last completed instance of a state*.

**SYNTAX**

```
end_tid([(QS)])
```

**PARAMETERS**

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

**RETURN TYPE**

integer

**SEE ALSO**

- “tid()” on page 11-24
- “start\_tid()” on page 11-47
- “offset\_tid()” on page 11-87

end\_cpu()

## DESCRIPTION

The `end_cpu()` function returns the logical CPU number associated with the *end event of the last completed instance of a state*. CPUs are logically numbered starting at 0 and monotonically increase thereafter.

## NOTE

This function is only valid when applied to events from Night-Trace kernel trace event files.

## SYNTAX

```
end_cpu [[QS]]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “cpu()” on page 11-25
- “start\_cpu()” on page 11-48
- “offset\_cpu()” on page 11-88



`end_offset()`**DESCRIPTION**

The `end_offset()` function returns the ordinal number (*offset*) of the *end event* of the *last completed instance of a state*.

**SYNTAX**

```
end_offset [(QS)]
```

**PARAMETERS**

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

**RETURN TYPE**

integer

**SEE ALSO**

- “offset()” on page 11-26
- “start\_offset()” on page 11-49

end\_time()

## DESCRIPTION

The `end_time()` function returns the time, in seconds, associated with the *end event* of the *last completed instance of a state*. Times are relative to the earliest trace event from all trace data files currently in use.

## SYNTAX

```
end_time [(QS)]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

double-precision floating point

## SEE ALSO

- “time()” on page 11-27
- “start\_time()” on page 11-50
- “state\_gap()” on page 11-73
- “state\_dur()” on page 11-74
- “offset\_time()” on page 11-89

`end_node_id()`**DESCRIPTION**

The `end_node_id()` function returns the internally-assigned *node identifier* associated with the *end event* of the *last completed instance of a state*.

**SYNTAX**

```
end_node_id [[QS]]
```

**PARAMETERS**

<i>QS</i>	A user-defined <i>qualified state</i> . If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.
-----------	---

**RETURN TYPE**

integer

**SEE ALSO**

- “`node_id()`” on page 11-28
- “`start_node_id()`” on page 11-51
- “`offset_node_id()`” on page 11-90

end\_pid\_table\_name()

## DESCRIPTION

The `end_pid_table_name()` function returns the name of the internally-assigned NightTrace process identifier table (*PID table*) associated with the *end event* of the *last completed instance of a state*.

## SYNTAX

```
end_pid_table_name [(QS)]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

string

## SEE ALSO

- “`pid_table_name()`” on page 11-29
- “`start_pid_table_name()`” on page 11-52
- “`offset_pid_table_name()`” on page 11-91

`end_tid_table_name()`

## DESCRIPTION

The `end_tid_table_name()` function returns the name of the internally-assigned NightTrace thread identifier table (*TID table*) associated with the *end event* of the *last completed instance of a state*.

## SYNTAX

```
end_tid_table_name [(QS)]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

string

## SEE ALSO

- “`tid_table_name()`” on page 11-30
- “`start_tid_table_name()`” on page 11-53
- “`offset_tid_table_name()`” on page 11-92

end\_node\_name()

## DESCRIPTION

The `end_node_name()` function returns the name of the system from which the *end event* of the *last completed instance of a state* was logged.

## SYNTAX

```
end_node_name [(QS)]
```

## PARAMETERS

<i>QS</i>	A user-defined <i>qualified state</i> . If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.
-----------	---

## RETURN TYPE

string

## SEE ALSO

- “node\_name()” on page 11-31
- “start\_node\_name()” on page 11-54
- “offset\_node\_name()” on page 11-93

## Multi-State Functions

Multi-state functions return information about one or more instances of a state:

- `state_gap()`
- `state_dur()`
- `state_matches()`
- `state_status()`

For restrictions on usage, see “State Graph” on page 10-32.

`state_gap()`

### DESCRIPTION

The `state_gap()` function returns the time in seconds between the *start event* of the *most recent instance of the state* and the *end event* of the instance immediately preceding it or zero if there was no previous instance.

### SYNTAX

```
state_gap [[QS]]
```

### PARAMETERS

*QS*                      A user-defined *qualified state*. If supplied, it specifies the *state* to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

### RETURN TYPE

double-precision floating point

### SEE ALSO

- “`start_time()`” on page 11-50
- “`end_time()`” on page 11-68
- “`event_gap()`” on page 11-35
- “`state_dur()`” on page 11-74

state\_dur()

## DESCRIPTION

The `state_dur()` function returns the time in seconds between the *start event* and the *end event* of the *last completed instance of a state*. Thus, if the *current time line* occurs within an instance of the state but before it has ended, `state_dur()` returns the duration of the previous instance or zero if there was no previous instance.

## SYNTAX

```
state_dur [[QS]]
```

## PARAMETERS

<i>QS</i>	A user-defined <i>qualified state</i> . If supplied, it specifies the <i>state</i> to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.
-----------	--

## RETURN TYPE

double-precision floating point

## SEE ALSO

- “state\_gap()” on page 11-73



state\_matches()

## DESCRIPTION

The `state_matches()` function returns the number of completed instances of a state on or before the *current time line*.

## SYNTAX

```
state_matches [(QS)]
```

## PARAMETERS

*QS* A user-defined *qualified state*. If supplied, it specifies the *state* to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

integer

## SEE ALSO

- “Start Functions” on page 11-37
- “summary\_matches()” on page 11-103

state\_status()

## DESCRIPTION

The `state_status()` function indicates whether the *current time line* resides within a *current instance of a state*. Thus, if the current time line is positioned in the region from the *start event* up to, but not including, the *end event* of an instance of the state, the return value is `TRUE`. Otherwise, it is `FALSE`.

## SYNTAX

```
state_status [(QS)]
```

## PARAMETERS

*QS*

A user-defined *qualified state*. If supplied, it specifies the *state* to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 11-116.

## RETURN TYPE

boolean

## Offset Functions

All offset functions take an expression that evaluates to an ordinal trace event (*offset*) as a parameter. (Offsets begin at zero.) These functions include the following:

- `offset_id()`
- `offset_arg()`
- `offset_arg_dbl()`
- `offset_num_args()`
- `offset_pid()`
- `offset_raw_pid()`
- `offset_lwpid()`
- `offset_thread_id()`
- `offset_task_id()`
- `offset_tid()`
- `offset_cpu()`
- `offset_time()`
- `offset_node_id()`
- `offset_pid_table_name()`
- `offset_tid_table_name()`
- `offset_node_name()`
- `offset_process_name()`
- `offset_task_name()`
- `offset_thread_name()`

Usually, these functions take one of the following functions as a parameter:

- `offset()`
- `start_offset()`
- `end_offset()`
- `min_offset()`
- `max_offset()`

For information about these functions, see “`offset()`” on page 11-26, “`start_offset()`” on page 11-49, “`end_offset()`” on page 11-67, “`min_offset()`” on page 11-101, and “`max_offset()`” on page 11-102.

## offset\_id()

### DESCRIPTION

The `offset_id()` function returns the *trace event ID* of the ordinal trace event (*offset*).

### SYNTAX

```
offset_id( offset_expr )
```

### PARAMETERS

<i>offset_expr</i>	An expression that evaluates to the <i>offset</i> (or ordinal trace event number) of a trace event.
--------------------	---

### RETURN TYPE

integer

### SEE ALSO

- “id()” on page 11-15
- “start\_id()” on page 11-38
- “end\_id()” on page 11-56

**offset\_arg()****DESCRIPTION**

The `offset_arg()` function returns the value of a particular *trace event argument* for the ordinal trace event (*offset*).

**SYNTAX**

`offset_arg[N] (offset_expr)`

**PARAMETERS**

<i>N</i>	Specifies the <i>N</i> th argument logged with the trace event. Defaults to 1.
<i>offset_expr</i>	An expression that evaluates to the <i>offset</i> (or ordinal trace event number) of a trace event.

**RETURN TYPE**

integer

**SEE ALSO**

- “arg()” on page 11-16
- “start\_arg()” on page 11-39
- “end\_arg()” on page 11-57
- “offset\_arg\_dbl()” on page 11-80
- “offset\_num\_args()” on page 11-81

## offset\_arg\_dbl()

### DESCRIPTION

The `offset_arg_dbl()` function returns the value of a particular *trace event argument* for the ordinal trace event (*offset*).

### SYNTAX

```
offset_arg[N]_dbl (offset_expr)
```

### PARAMETERS

<i>N</i>	Specifies the <i>N</i> th argument logged with the trace event. Defaults to 1.
<i>offset_expr</i>	An expression that evaluates to the <i>offset</i> (or ordinal trace event number) of a trace event.

### RETURN TYPE

double-precision floating point

### SEE ALSO

- “arg\_dbl()” on page 11-17
- “start\_arg\_dbl()” on page 11-40
- “end\_arg\_dbl()” on page 11-58
- “offset\_arg()” on page 11-79
- “offset\_num\_args()” on page 11-81

**offset\_num\_args()****DESCRIPTION**

The `offset_num_args()` function returns the number of arguments logged with the ordinal trace event (*offset*).

**SYNTAX**

```
offset_num_args (offset_expr)
```

**PARAMETERS**

*offset\_expr*            An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

**RETURN TYPE**

integer

**SEE ALSO**

- “num\_args()” on page 11-18
- “start\_num\_args()” on page 11-41
- “end\_num\_args()” on page 11-59
- “offset\_arg()” on page 11-79
- “offset\_arg\_dbl()” on page 11-80

## offset\_pid()

### DESCRIPTION

The `offset_pid()` function returns the PID from which the ordinal trace event (*offset*) was logged.

### NOTE

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. See the `_lwp_global_self(2)` man page for more information.

On Linux systems, the `offset_pid()` returns the operating system process identifier.

### SYNTAX

```
offset_pid(offset_expr)
```

### PARAMETERS

*offset\_expr*            An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

### RETURN TYPE

integer

### SEE ALSO

- “pid()” on page 11-19
- “raw\_pid()” on page 11-20
- “lwpid()” on page 11-21
- “start\_pid()” on page 11-42
- “end\_pid()” on page 11-60



**offset\_raw\_pid()****DESCRIPTION**

The `offset_raw_pid()` function returns the process identifier (*raw PID*) from which the ordinal trace event (*offset*) was logged.

**NOTE**

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. The `offset_raw_pid()` function returns the upper 16 bits. See the `getpid(2)` man page for more information.

On Linux systems, the `offset_raw_pid()` returns the operating system process identifier.

**SYNTAX**

```
offset_raw_pid(offset_expr)
```

**PARAMETERS**

<i>offset_expr</i>	An expression that evaluates to the <i>offset</i> (or ordinal trace event number) of a trace event.
--------------------	---

**RETURN TYPE**

integer

**SEE ALSO**

- “pid()” on page 11-19
- “raw\_pid()” on page 11-20
- “lwpid()” on page 11-21
- “start\_pid()” on page 11-42
- “end\_pid()” on page 11-60

## offset\_lwpid()

### DESCRIPTION

The `offset_lwpid()` function returns the lightweight process identifier (*LWPID*) from which the ordinal trace event (*offset*) was logged.

### NOTE

On PowerMAX OS systems, a PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. The `offset_lwpid()` function returns the lower 16 bits. See the `_lwp_self(2)` man page for more information.

On Linux systems, `offset_lwpid()` returns the operating system process identifier.

### SYNTAX

```
offset_lwpid(offset_expr)
```

### PARAMETERS

*offset\_expr*      An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

### RETURN TYPE

integer

### SEE ALSO

- “pid()” on page 11-19
- “raw\_pid()” on page 11-20
- “lwpid()” on page 11-21
- “start\_lwpid()” on page 11-44
- “end\_lwpid()” on page 11-62

**offset\_thread\_id()****DESCRIPTION**

The `offset_thread_id()` function returns the *thread* identifier from which the ordinal trace event (*offset*) was logged.

**NOTE**

See the **thr\_self(3thread)** man page for more information.

**SYNTAX**

`offset_thread_id(offset_expr)`

**PARAMETERS**

*offset\_expr*      An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

**RETURN TYPE**

integer

**SEE ALSO**

- “`thread_id()`” on page 11-22
- “`start_thread_id()`” on page 11-45
- “`end_thread_id()`” on page 11-63

## **offset\_task\_id()**

### **DESCRIPTION**

The `offset_task_id()` function returns the Ada task identifier from which the ordinal trace event (*offset*) was logged.

### **NOTE**

This function is only meaningful for trace events logged by Ada tasking programs.

### **SYNTAX**

`offset_task_id(offset_expr)`

### **PARAMETERS**

*offset\_expr*            An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

### **RETURN TYPE**

integer

### **SEE ALSO**

- “`task_id()`” on page 11-23
- “`start_task_id()`” on page 11-46
- “`end_task_id()`” on page 11-64

**offset\_tid()****DESCRIPTION**

The `offset_tid()` function returns the internally-assigned NightTrace thread identifier (*TID*) from which the ordinal trace event (*offset*) was logged.

**SYNTAX**

```
offset_tid(offset_expr)
```

**PARAMETERS**

<i>offset_expr</i>	An expression that evaluates to the <i>offset</i> (or ordinal trace event number) of a trace event.
--------------------	---

**RETURN TYPE**

integer

**SEE ALSO**

- “tid()” on page 11-24
- “start\_tid()” on page 11-47
- “end\_tid()” on page 11-65

## offset\_cpu()

### DESCRIPTION

The `offset_cpu()` function returns the logical CPU number on which the ordinal trace event (*offset*) occurred. CPUs are logically numbered starting at 0 and monotonically increase thereafter.

### NOTE

This function is only valid when applied to events from Night-Trace kernel trace event files.

### SYNTAX

```
offset_cpu(offset_expr)
```

### PARAMETERS

*offset\_expr*            An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

### RETURN TYPE

integer

### SEE ALSO

- “cpu()” on page 11-25
- “start\_cpu()” on page 11-48
- “end\_cpu()” on page 11-66

**offset\_time()****DESCRIPTION**

The `offset_time()` function returns the time in seconds between the beginning of the trace run and the ordinal trace event (*offset*).

**SYNTAX**

`offset_time(offset_expr)`

**PARAMETERS**

*offset\_expr*      An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

**RETURN TYPE**

double-precision floating point

**SEE ALSO**

- “time()” on page 11-27
- “start\_time()” on page 11-50
- “end\_time()” on page 11-68

## offset\_node\_id()

### DESCRIPTION

The `offset_node_id()` function returns the internally-assigned *node identifier* from which the ordinal trace event (*offset*) was logged.

### SYNTAX

```
offset_node_id(offset_expr)
```

### PARAMETERS

<i>offset_expr</i>	An expression that evaluates to the <i>offset</i> (or ordinal trace event number) of a trace event.
--------------------	---

### RETURN TYPE

integer

### SEE ALSO

- “node\_id()” on page 11-28
- “start\_node\_id()” on page 11-51
- “end\_node\_id()” on page 11-69



**offset\_pid\_table\_name()****DESCRIPTION**

The `offset_pid_table_name()` function returns the name of the internally-assigned NightTrace process identifier table (*PID table*) for the ordinal trace event (*offset*).

**SYNTAX**

```
offset_pid_table_name(offset_expr)
```

**PARAMETERS**

*offset\_expr*            An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

**RETURN TYPE**

string

**SEE ALSO**

- “`pid_table_name()`” on page 11-29
- “`start_pid_table_name()`” on page 11-52
- “`end_pid_table_name()`” on page 11-70

## **offset\_tid\_table\_name()**

### **DESCRIPTION**

The `offset_tid_table_name()` function returns the name of the internally-assigned NightTrace thread identifier table (*TID table*) for the ordinal trace event (*offset*).

### **SYNTAX**

```
offset_tid_table_name(offset_expr)
```

### **PARAMETERS**

*offset\_expr*            An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

### **RETURN TYPE**

string

### **SEE ALSO**

- “tid\_table\_name()” on page 11-30
- “start\_tid\_table\_name()” on page 11-53
- “end\_tid\_table\_name()” on page 11-71

**offset\_node\_name()****DESCRIPTION**

The `offset_node_name()` function returns the name of the system from which the ordinal trace event (*offset*) was logged.

**SYNTAX**

`offset_node_name (offset_expr)`

**PARAMETERS**

*offset\_expr*            An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

**RETURN TYPE**

string

**SEE ALSO**

- “node\_name()” on page 11-31
- “start\_node\_name()” on page 11-54
- “end\_node\_name()” on page 11-72

## **offset\_process\_name()**

### **DESCRIPTION**

The `offset_process_name()` function returns the name of the process (*PID*) from which the ordinal trace event (*offset*) was logged.

### **SYNTAX**

```
offset_process_name(offset_expr)
```

### **PARAMETERS**

<i>offset_expr</i>	An expression that evaluates to the <i>offset</i> (or ordinal trace event number) of a trace event.
--------------------	---

### **RETURN TYPE**

string

### **SEE ALSO**

- “`process_name()`” on page 11-32

## **offset\_task\_name()**

### **DESCRIPTION**

The `offset_task_name()` function returns the name of the task from which the ordinal trace event (*offset*) was logged.

### **NOTE**

This function is only meaningful for trace events which were logged from Ada tasking programs.

### **SYNTAX**

`offset_task_name (offset_expr)`

### **PARAMETERS**

*offset\_expr*            An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

### **RETURN TYPE**

string

### **SEE ALSO**

- “task\_name()” on page 11-33

## **offset\_thread\_name()**

### **DESCRIPTION**

The `offset_thread_name()` function returns the thread name from which the ordinal trace event (*offset*) was logged.

### **SYNTAX**

```
offset_thread_name(offset_expr)
```

### **PARAMETERS**

<i>offset_expr</i>	An expression that evaluates to the <i>offset</i> (or ordinal trace event number) of a trace event.
--------------------	---

### **RETURN TYPE**

string

### **SEE ALSO**

- “`thread_name()`” on page 11-34

## Summary Functions

You usually use summary functions on the **Summarize Form**. Except for `summary_matches()`, all of these functions take another expression as a parameter. They include the following:

- `min()`
- `max()`
- `avg()`
- `sum()`
- `min_offset()`
- `max_offset()`
- `summary_matches()`

### **min()**

#### **DESCRIPTION**

The `min()` function returns the minimum value of all occurrences of *expr* within a time range. When used in a **Summarize Form**, the time range is defined by that form. When used elsewhere, the time range is defined as the region starting with the first *trace event* and ending with the *current trace event*.

#### **SYNTAX**

`min(expr)`

#### **PARAMETERS**

*expr*                      A numeric expression.

#### **RETURN TYPE**

data type of *expr*

#### **SEE ALSO**

- “Summary Functions” on page 11-97
- “Summarizing Statistical Information” on page 12-12

## max()

### DESCRIPTION

The `max()` function returns the maximum value of all occurrences of *expr* within a time range. When used in a **Summarize Form**, the time range is defined by that form. When used elsewhere, the time range is defined as the region starting with the first *trace event* and ending with the *current trace event*.

### SYNTAX

`max(expr)`

### PARAMETERS

*expr*                      A numeric expression.

### RETURN TYPE

data type of *expr*

### SEE ALSO

- “Summary Functions” on page 11-97
- “Summarizing Statistical Information” on page 12-12



**avg()****DESCRIPTION**

The `avg()` function returns the average value of all occurrences of *expr* within a time range. When used in a **Summarize Form**, the time range is defined by that form. When used elsewhere, the time range is defined as the region starting with the first *trace event* and ending with the *current trace event*.

**SYNTAX**

`avg(expr)`

**PARAMETERS**

*expr*                      A numeric expression.

**RETURN TYPE**

data type of *expr*

**SEE ALSO**

- “Summary Functions” on page 11-97
- “Summarizing Statistical Information” on page 12-12

## sum()

### DESCRIPTION

The `sum()` function returns the sum value of all occurrences of *expr* within a time range. When used in a **Summarize Form**, the time range is defined by that form. When used elsewhere, the time range is defined as the region starting with the first *trace event* and ending with the *current trace event*.

### SYNTAX

`sum(expr)`

### PARAMETERS

*expr*                      A numeric expression.

### RETURN TYPE

data type of *expr*

### SEE ALSO

- “Summary Functions” on page 11-97
- “Summarizing Statistical Information” on page 12-12

**min\_offset()****DESCRIPTION**

The `min_offset()` function returns the ordinal trace event (*offset*) where the minimum value of the parameter occurred for matches in the time range. Thus, if the same minimum was seen more than once, the offset corresponds to the first one seen.

**SYNTAX**

```
min_offset(expr)
```

**PARAMETERS**

*expr*                    A numeric expression.

**RETURN TYPE**

integer

**NOTE**

There is no function that returns the trace event ID where the minimum value of the first argument occurred for all matches in the time range. You could obtain this value by nesting the functions as follows:

```
offset_id( min_offset( arg1() ) )
```

**SEE ALSO**

- “Summary Functions” on page 11-97
- “Summarizing Statistical Information” on page 12-12

## max\_offset()

### DESCRIPTION

The `max_offset()` function returns the ordinal trace event (*offset*) where the maximum value of the parameter occurred for matches in the time range. Thus, if the same maximum was seen more than once, the offset corresponds to the first one seen.

### SYNTAX

```
max_offset(expr)
```

### PARAMETERS

*expr*                    A numeric expression.

### RETURN TYPE

integer

### NOTE

There is no function that returns the trace event ID where the maximum value of the first argument occurred for all matches in the time range. You could obtain this value by nesting the functions as follows:

```
offset_id( max_offset( arg1() ) )
```

### SEE ALSO

- “Summary Functions” on page 11-97
- “Summarizing Statistical Information” on page 12-12

## summary\_matches()

### DESCRIPTION

The `summary_matches()` function returns the number of times the summary criteria was matched in the time range.

### NOTE

This function should only be used in the **Summarize NightTrace Events** dialog. Its behavior elsewhere is undefined. (See “Summarizing Statistical Information” on page 12-12 for more information.)

### SYNTAX

```
summary_matches()
```

### RETURN TYPE

integer

### SEE ALSO

- “`event_matches()`” on page 11-36
- “`state_matches()`” on page 11-75

## Format and Table Functions

The format function allows you to display a string. The table functions allow you to extract information from user-defined and pre-defined string and format tables. These functions include the following:

- `get_string()`
- `get_item()`
- `get_format()`
- `format()`

For more information about tables, see “Tables” on page 4-13 and “Kernel String Tables” on page 13-13.

### **get\_string()**

The `get_string()` routine dynamically looks up a string in a string table.

#### **SYNTAX**

```
get_string(table_name[, int_expr])
```

#### **PARAMETERS**

*table\_name*            *table\_name* is an unquoted character string that represents the name of a string table. To avoid possible forward reference problems, try to make your `get_string()` calls refer to previously-defined string tables. The following string table names are pre-defined in NightTrace: `event`, `pid`, `tid`, `boolean`, `name_pid`, `name_tid`, `node_name`, `pid_nodename`, `tid_nodename`, `vector`, `syscall`, `device`, `event_summary`, `event_arg_summary`, `event_arg_dbl_summary`, `state_summary`. For more information on these tables, see “Pre-Defined String s” on page 4-16 and “Kernel String Tables” on page 13-13.

*int\_expr*            *int\_expr* is an integer expression that acts as an index into the specified string table. *int\_expr* must either match an identifying integer value in the *table\_name* string table, or the *table\_name* string table must have a default item line; otherwise `get_string()` returns a string of *int\_expr* in decimal. Often *int\_expr* is based on a NightTrace function.

If your table consists of only a default item line, omit this parameter.

## DESCRIPTION

The following NightTrace constructs can call `get_string()` to dynamically locate a static string in a string table:

- A Then-Expression of a display object configuration
- A value field of a format table

For each `get_string()` call, NightTrace follows these steps:

1. Evaluates *int\_expr*
2. Uses this value as an index into *table\_name*
3. Retrieves the associated string from *table\_name*
4. Returns a string

The following lines provide a brief example of a call to `get_string()`.

```
string_table (conditions) = {
    item = 1, "normal";
    item = 50, "YELLOW ALERT";
    item = 99, "RED ALERT";
    default_item = "N/A";
};
```

In this example the numeric argument associated with a trace event represents the current conditions (*conditions*). If the argument has the value 99, NightTrace:

1. Uses the value 99 as in index into *conditions*
2. Retrieves the associated string ("RED ALERT") from *conditions*
3. Returns "RED ALERT"

## RETURN TYPES

On successful completion, `get_string()` returns a string from a string table. NightTrace returns a string of the item number, *int\_expr*, in decimal if *table\_name* is not found, or if *int\_expr* is not found and there is no default item line. The first time *table\_name* is not found, NightTrace issues an error message. Because `get_string()` returns a string, you can use it anywhere a string expression is appropriate.

For more information on string tables, see "String Tables" on page 4-15.

**get\_item()**

The `get_item()` routine looks up an item number in a string table.

**SYNTAX**

```
int get_item(table_name, "str_const")
```

**PARAMETERS**

*table\_name*            *table\_name* is an unquoted character string that represents the name of a string table. To avoid possible forward reference problems, try to make your `get_item()` calls refer to previously-defined string tables. The following string table names are pre-defined in NightTrace: `event`, `pid`, `tid`, `boolean`, `name_pid`, `name_tid`, `node_name`, `pid_nodename`, `tid_nodename`, `vector`, `syscall`, `device`, `event_summary`, `event_arg_summary`, `event_arg_dbl_summary`, `state_summary`. For more information on these tables, see "Pre-Defined String s" on page 4-16 and "Kernel String Tables" on page 13-13.

*str\_const*            *str\_const* is a string constant literal that acts as an index into the specified string table. *str\_const* must either exactly match a string value in the *table\_name* string table, or the *table\_name* string table must have a default item line; otherwise the results are undefined. A *table\_name* may contain several item lines with the same *str\_const* value.

**DESCRIPTION**

Typically, a `get_item()` call is used in conditional expressions for qualified expressions, searches, summaries, or display object configurations.

The `get_item()` call returns an index number into the specified string table (*table\_name*) for the first item in the table which matches the specified string (*str\_const*).

For example, assume that the following string table definition is in your page configuration file (see "String Tables" on page 4-15):

```
string_table (fruit) = {
    item = 3, "apple";
    item = 4, "orange";
    item = 5, "cherry";
    item = 6, "banana";
    default_item = "Unknown";
};
```

A `get_item()` call can be used in an **If Condition** when configuring a Data Box (see "Data Box" on page 10-19):

```
If Condition        arg1 = get_item(fruit, "cherry")
```



requiring the first argument of the associated trace event to be the same as the index value matching the entry for `cherry` in the `fruit` string table (which, in our example, is 5).

## RETURN TYPES

On successful completion, `get_item()` returns an item number from a string table. If several item lines within the string table have the same string value as `str_const`, `get_item()` returns the first item number from one of these item lines. If `table_name` is not found, NightTrace issues an error message, and the results are undefined. If `str_const` is not found and there is no default item line, the results are undefined. Because `get_item()` returns an integer, you can use it anywhere an integer expression can be used.

For more information on string tables, see “String Tables” on page 4-15.

## get\_format()

The `get_format()` routine dynamically looks up a string in a format table.

### SYNTAX

```
get_format (table_name[, int_expr])
```

### PARAMETERS

*table\_name*            *table\_name* is an unquoted character string that represents the name of a format table. To avoid possible forward reference problems, try to make your `get_format()` calls refer to previously-defined format tables.

*int\_expr*            *int\_expr* is an integer expression that acts as an index into the specified format table. *int\_expr* must either match an identifying integer value in the *table\_name* format table, or the *table\_name* format table must have a default item line; otherwise, the results are undefined. Often *int\_expr* is based on a NightTrace function.

If your table consists of only a default item line, omit this parameter.

### DESCRIPTION

A call to `get_format()` must be the first function call in an expression. You must not nest calls to `get_format()`.

The **Then-Expression** parameter of a Data Box configuration and the **Summarize-Expression** on a Summary Form can call `get_format()` to dynamically locate a string in a format table. For each `get_format()` call, NightTrace follows these steps:

1. Evaluates *int\_expr*
2. Uses this value as an index into *table\_name*
3. Retrieves the associated string from *table\_name*
4. Replaces any conversion specifications in the associated string
5. Returns a string

Assume that the following format table definition is in your configuration file.

```
format_table (what_pid) = {
    item = 1, "Trace event 1 logged by pid %d'%d", "raw_pid()",
           "lwpid()";
    default_item = "Unaccounted for event ID (%d)", "id()";
};
```

Assume that you make the following call in the **Then-Expression** of a Data Box.

```
get_format (what_pid, id())
```

In this example, the `what_pid` format table associates one dynamically-generated string with trace event ID 1 (`id() == 1`) and another string with all other trace events (`default_item`). When `NightTrace` processes a trace event for the display object with the above `get_format()`, it:

1. Evaluates the `NightTrace id()` function. (Assume it evaluates to 1)
2. Calls `get_format()`
3. Uses this value (1) as an index into the `what_pid` format table
4. Retrieves the associated string ("Trace event 1 logged by pid %d' %d") from the `what_pid` format table
5. Evaluates the `NightTrace raw_pid()` and `lwpid()` functions. (Assume they evaluate to 213 and 1 respectively)
6. Replaces the `%d` conversion specifiers with the `raw_pid()` and `lwpid()` values
7. Displays "Trace event 1 logged by pid 213'1"

## RETURN TYPES

On successful completion, `get_format()` returns a format table string. Otherwise, it returns an empty string.

For more information on format tables, see "Format Tables" on page 4-19.

## format()

The `format()` routine displays a string.

### SYNTAX

```
format ("format_string" [, arg] ...)
```

### PARAMETERS

*format\_string*      *format\_string* controls how the optional *args* are displayed. *format\_string* is based on the format parameter used in the **printf(3S)** routine in C. It is a character string enclosed in double quotes that contains literal characters and conversion specifications. The literals are copied as is to the display object. Conversion specifications modify zero or more *args*.

*arg*                      *arg* is an optional expression to be formatted and displayed.

### DESCRIPTION

Call the `format()` function to display a string. You can do this only from the **Then-Expression** parameter of a display object configuration or the **Summary-Expression** of the **Summarize Form**. A call to `format()` must be the first function call in an expression. You must not nest calls to `format()`.

The following lines provide examples of `format()` statements and what they display. Assume all variables have a value of 10 (decimal).

```
format( "Error")                      Error
format( "Event=%d", id() )            Event=10
format( "Argument is %X", arg1())    Argument is A
```

### RETURN TYPES

On successful completion, `format()` returns a string. Otherwise, it returns an empty string.

## Macros

*Macros* are user-defined expressions provided for flexibility and convenience. Unlike functions, they do not have any parameters. They are invoked by calling the macro with a `$` before the macro name.

To create a macro definition:

- select the **Expressions...** menu item from the **Edit** menu on any display page (see “Edit” on page 9-5)
- press the **Add...** button on the **NightTrace Qualified Expressions** dialog (see “NightTrace Qualified Expressions” on page 11-119)
- choose **Macro** from the **Key** drop-down menu in the **Edit NightTrace Qualified Expression** dialog (see “Edit NightTrace Qualified Expression” on page 11-122)
- enter a valid expression in the **Expansion** field (see “Expressions” on page 11-1)
- assign a **Name** to this macro (or use the name generated by NightTrace)
- press the **OK** button

### NOTE

Macros cannot be called recursively; if attempted, the results are undefined and NightTrace will issue an error.

Furthermore, macros must not call the `format()` and `get_format()` functions (see “format()” on page 11-110 and “get\_format()” on page 11-108).

For example, a macro is useful when configuring a State Graph (see “State Graph” on page 10-32) because it has two expressions that are often related. Consider the following configuration:

<b>Start Events</b>	FOO
<b>Start Condition</b>	<code>arg1() == 0x1234 &amp;&amp; (arg2() == 0    arg3() &gt; 700)</code>
<b>End Events</b>	BAR
<b>End Condition</b>	<code>arg1() == 0x1234 &amp;&amp; (arg2() == 0    arg3() &gt; 700)</code>

This display object graphs states of trace event FOO through trace event BAR, where the arguments of the trace events must meet an identical criteria to be considered interesting.

Since the same expression:

```
arg1() == 0x1234 && (arg2() == 0 || arg3() > 700)
```

is used in both the **Start Condition** and the **End Condition**, it is an ideal candidate for a macro.

Therefore, a macro definition of:

<b>Name</b>	foobar
<b>Expansion</b>	arg1() == 0x1234 && (arg2() == 0    arg3() > 700)

would allow our earlier State Graph configuration to be defined as:

<b>Start Events</b>	FOO
<b>Start Condition</b>	\$foobar
<b>End Events</b>	BAR
<b>End Condition</b>	\$foobar

Using the macro name in place of the expression reduces possible errors that might occur when manually entering the expression. In addition, using a macro provides the flexibility of changing the expression in the macro definition and having those changes propagated wherever the macro is used.

## Qualified Events

*Qualified events* provide a means for referencing a set of one or more trace events which may be restricted by conditions specified by the user.

Qualified events can be used within trace event functions (see “Trace Event Functions” on page 11-14).

To create a qualified event definition:

- select the **Expressions...** menu item from the **Edit** menu on any display page (see “Edit” on page 9-5)
- press the **Add...** button on the **NightTrace Qualified Expressions** dialog (see “NightTrace Qualified Expressions” on page 11-119)
- choose **Event** from the **Key** drop-down menu in the **Edit NightTrace Qualified Expression** dialog (see “Edit NightTrace Qualified Expression” on page 11-122)

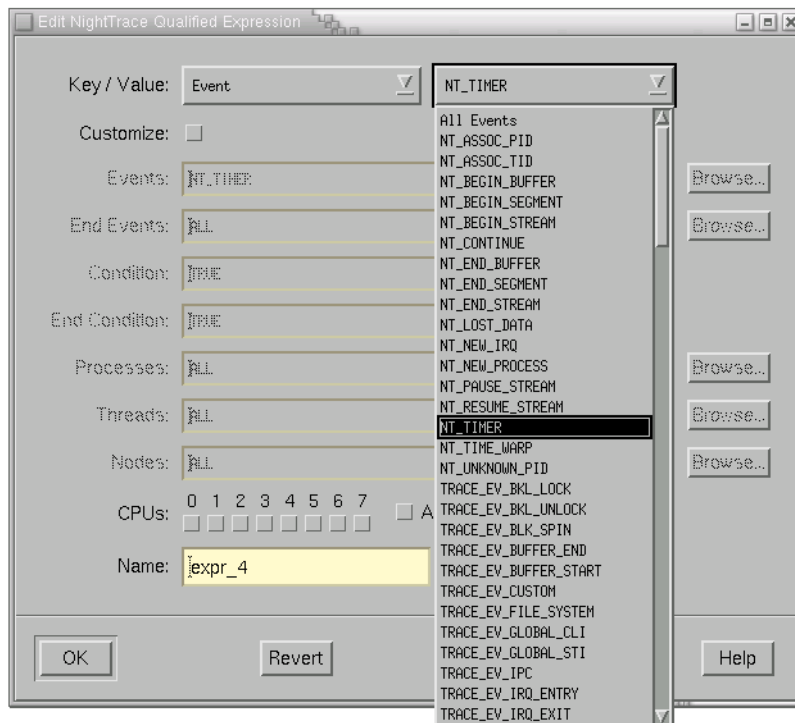
### NOTE

You may also select any of the predefined qualified events provided in the **Key** drop-down menu (e.g. *System Call Events*, *Interrupt Leave*, *Exception*, etc.) and further customize them to your specifications. See “Key / Value” on page 11-123 for a description of the predefined qualified events in the **Key** drop-down menu.

- select the desired event from the **Value** drop-down menu

### NOTE

The event selected in the **Value** drop-down menu will be added to the **Events** field.



**Figure 11-3. Choosing a Key / Value pair for a qualified event**

- check the **Customize** button to configure the qualified expression to your specifications assigning values to **Events**, **Condition**, **Processes**, **Threads**, **Nodes**, and **CPUs** as appropriate
- assign a **Name** to this qualified event (or use the name generated by NightTrace)
- press the **OK** button

**TIP**

Consider giving your trace events uppercase names in event-map files (see “Event Map Files” on page 4-10) and giving any corresponding qualified event the same name in lowercase.

Qualified events can be useful when you are interested in seeing a trace event (or state) that occurs within a certain amount of time after another trace event.



For example, given the following qualified event configuration:

<b>Name</b>	<code>fire</code>
<b>Events</b>	<code>FIRE</code>
<b>CPUs</b>	<code>2</code>

an Event Graph (see “Event Graph” on page 10-26) can be configured to show only `BAR` trace events that happen within 100 microseconds of a `FIRE` trace event on CPU 2:

<b>Events</b>	<code>BAR</code>
<b>If Condition</b>	<code>time() - time(fire) &lt; 100us</code>

Note that although the `BAR` trace events themselves can happen on any CPU, they will be graphed only if they occur within 100 microseconds of a `FIRE` trace event on CPU 2 (see “time()” on page 11-27).

## Qualified States

*Qualified states* provide a means for defining regions of time based on specific starting and ending events and restricted by conditions specified by the user.

Qualified states can be used to reference user-defined regions of time within start functions (see “Start Functions” on page 11-37), end functions (see “End Functions” on page 11-55), and multi-state functions (see “Multi-State Functions” on page 11-73).

To create a qualified state definition:

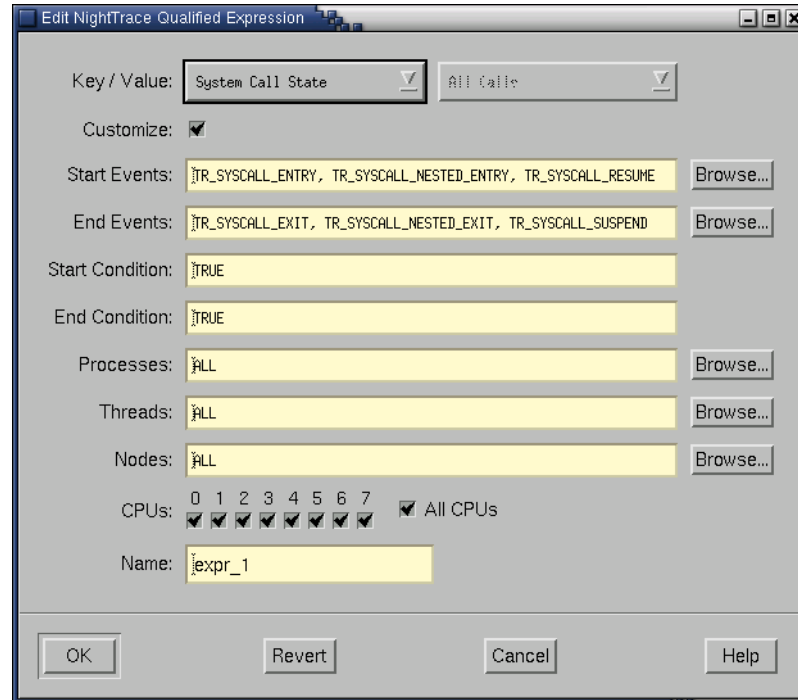
- select the **Expressions...** menu item from the **Edit** menu on any display page (see “Edit” on page 9-5)
- press the **Add...** button on the **NightTrace Qualified Expressions** dialog (see “NightTrace Qualified Expressions” on page 11-119)
- choose **State** from the **Key** drop-down menu in the **Edit NightTrace Qualified Expression** dialog (see “Edit NightTrace Qualified Expression” on page 11-122)

### NOTE

You may also create a qualified state using a previously-defined qualified state as a basis by choosing **Qualified State** from the **Key** drop-down menu and selecting the desired item from the **Value** drop-down menu. (Only qualified states previously defined in the current session will appear in the **Value** list.)

Furthermore, you may select any of the predefined qualified states provided in the **Key** drop-down menu (e.g. **System Call State**, **Interrupt State**, **Exception State**) and further customize them to your specifications. See “Key / Value” on page 11-123 for a description of the predefined qualified states in the **Key** list.

- check the **Customize** button (if necessary) to configure the qualified expression to your specifications assigning values to **Start Events**, **End Events**, **Start Condition**, **End Condition**, **Processes**, **Threads**, **Nodes**, and **CPUs** as appropriate



**Figure 11-4. Customizing a qualified state**

- assign a **Name** to this qualified state (or use the name generated by Night-Trace)
- press the OK button

Qualified states can be useful when you are interested in a trace event that occurs while a certain state is active. For example, the following qualified state:

```
Name           foo_state
Start Events    PROG_A_BEGIN
End Events      PROG_A_EXIT
```

defines a state that is active whenever program A is running. Assume that another process is logging FOO trace events asynchronously. If you are interested only in the FOO trace events that are logged while program A is running, you can define an Event Graph (see “Event Graph” on page 10-26) as follows:

```
Events          FOO
If Condition     state_status(foo_state) == true
```

This will graph only FOO trace events that occur while the qualified state `foo_state` is active (see “`state_status()`” on page 11-76). Thus, you see only FOO trace events logged while program A is running.

**NOTE**

The “== true” in the If Expression is not necessary since the type of value returned from the `state_status()` call is boolean.

## NightTrace Qualified Expressions

The NightTrace Qualified Expressions dialog allows the user to define new qualified expressions (*qualified events*, *qualified states*, and *macros*) in the current session as well as edit existing qualified expressions (see “Qualified Events” on page 11-113, “Qualified States” on page 11-116, and “Macros” on page 11-111).

The NightTrace Qualified Expressions dialog presents an alphabetical list of all qualified expressions defined in the current session. Qualified expressions are global to all the display pages in the current session; that is, if a qualified expression is created by one display page, it may be used by any other display page.

The NightTrace Qualified Expressions dialog is presented when the Expressions... menu item is selected from the Edit menu of a display page (see “Expressions...” on page 9-7).

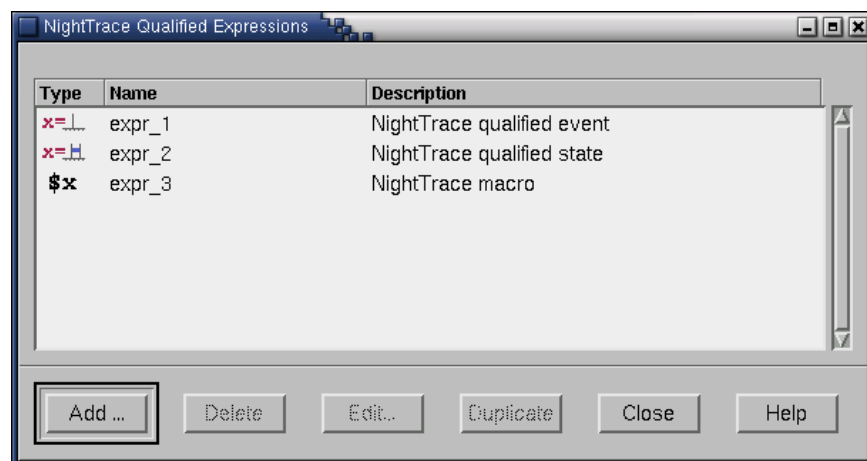


Figure 11-5. NightTrace Qualified Expressions dialog

### Type

This column displays an icon representing the type of qualified expression in the list.

---

	indicates a qualified event
	indicates a qualified state
	indicates a macro

---

### **Name**

The name to reference this qualified expression. This value is defined in the **Name** field of the **Edit NightTrace Qualified Expression** dialog (see “Edit NightTrace Qualified Expression” on page 11-122).

### **Description**

This column contains one of the following descriptions:

- NightTrace qualified event
- NightTrace qualified state
- NightTrace macro

depending on the type of qualified expression in the list.

The following buttons appear at the bottom of the **NightTrace Qualified Expressions** dialog and have the specified meaning:

### **Add...**

Presents the **Edit NightTrace Qualified Expression** dialog (see “Edit NightTrace Qualified Expression” on page 11-122) allowing the user to create a new qualified event, qualified state, or macro.

Once added, the new qualified expression will appear in the **NightTrace Qualified Expressions** dialog.

### **Delete**

Deletes the expression(s) selected in the **NightTrace Qualified Expressions** dialog from the current session.

### **Edit...**

Presents the **Edit NightTrace Qualified Expression** dialog (see “Edit NightTrace Qualified Expression” on page 11-122) allowing the user to edit the qualified expression selected in the **NightTrace Qualified Expressions** dialog.

### **Duplicate**

Creates a copy of the expression(s) selected in the **NightTrace Qualified Expressions** dialog and adds them to the list presented in this dialog.

The new expression(s) will be named

*orig\_copyn*

where *orig* is the name of the duplicated expression and *n* is an integer value.

**Close**

Closes the NightTrace Qualified Expressions dialog.

**Help**

Opens the HyperHelp viewer to the online help topic for this dialog.

Double-clicking on any item in the list will open the **Edit NightTrace Qualified Expression** dialog (see “Edit NightTrace Qualified Expression” on page 11-122) allowing the user to edit that particular qualified expression.

## Edit NightTrace Qualified Expression

The Edit NightTrace Qualified Expression dialog allows the user to create new qualified expressions in the form of *qualified events*, *qualified states*, and *macros* (see “Qualified Events” on page 11-113, “Qualified States” on page 11-116, and “Macros” on page 11-111). In addition, the user can edit qualified expressions that have been previously defined in the current session.

The Edit NightTrace Qualified Expression dialog is presented when either the Add... or Edit... button is pressed on the NightTrace Qualified Expressions dialog (see “NightTrace Qualified Expressions” on page 11-119).

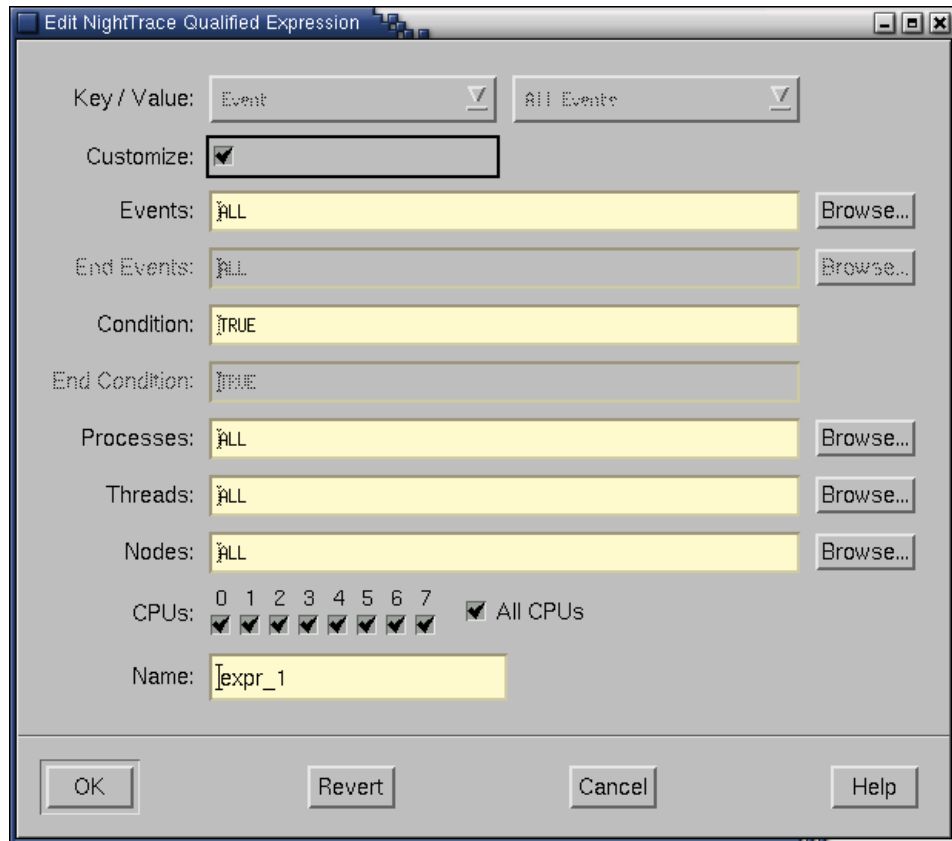


Figure 11-6. Edit NightTrace Qualified Expression dialog



**Key / Value**

These two drop-down menus are used in combination to provide a quick and efficient method for defining a NightTrace qualified expression.

The value chosen for the **Key** will determine the type of expression being created:

## Event

create a qualified event (see “Qualified Events” on page 11-113)

## State

create a qualified state (see “Qualified States” on page 11-116)

## Qualified State

create a qualified state using a previously-defined qualified state as a basis (see “Qualified States” on page 11-116)

## Macro

create a macro (see “Macros” on page 11-111)

## Process ID

create a qualified event restricted to selected processes

## Thread ID

create a qualified event restricted to certain threads

## System Call

create a qualified event for those instances when the kernel starts (or resumes) executing the source code for the particular system call selected in the **Value** drop-down menu

## System Call Leave

create a qualified event for those instances when the kernel exits (or suspends execution of) the particular system call selected in the **Value** drop-down menu

## System Call Events

create a qualified event for those instances when the kernel starts (or resumes) executing the source code for the particular system call selected in the **Value** drop-down menu as well as when the kernel exits (or suspends execution of) that particular system call

## System Call State

create a qualified state whose *start events* are defined as those instances when the kernel starts (or resumes) executing the source code for particular system call selected in the **Value** drop-down menu and whose *end*

*events* are comprised of those events when the kernel exits (or suspends execution of) that particular system call

#### Interrupt

create a qualified event for those instances when the kernel starts executing the source code for the particular interrupt selected in the **Value** drop-down menu

#### Interrupt Leave

create a qualified event for those instances when the kernel finishes executing the source code for the particular interrupt selected in the **Value** drop-down menu

#### Interrupt Events

create a qualified event for those instances when the kernel starts executing the source code for the particular interrupt selected in the **Value** drop-down menu as well as when the kernel exits that particular interrupt

#### Interrupt State

create a qualified state whose *start events* are defined as those instances when the kernel starts executing the source code for the particular interrupt selected in the **Value** drop-down menu and whose *end events* are comprised of those events when the kernel exits that particular interrupt

#### Exception

create a qualified event for those instances when the kernel starts (or resumes) executing the source code for the particular exception selected in the **Value** drop-down menu

#### Exception Leave

create a qualified event for those instances when the kernel exits (or suspends execution of) the particular exception selected in the **Value** drop-down menu

#### Exception Events

create a qualified event for those instances when the kernel starts (or resumes) executing the source code for the particular exception selected in the **Value** drop-down menu as well as when the kernel exits (or suspends execution of) that particular exception

#### Exception State

create a qualified state whose *start events* are defined as those instances when the kernel starts (or resumes) executing the source code for particular exception selected in the **Value** drop-down menu and whose *end events* are comprised of those events when the kernel exits (or suspends execution of) that particular exception

The **Value** drop-down menu provides a list of possible choices associated with the selected **Key**. The choices in this list are based on the trace dataset; for instance, if **Process ID** is selected for the **Key**, the **Value** drop-down menu will consist of those processes in the current dataset that logged trace events and/or those that were executing when kernel trace events were collected.

The qualified expression can be further customized by selecting the **Customize** checkbox (see below) and modifying the remaining fields.

### Customize

Use this option to further configure your qualified expression.

When **Customize** is checked, the **Key / Value** drop-down menus become disabled (desensitized). The remaining fields (which are populated with values corresponding to the **Key / Value** pair) may be modified to tailor the qualified expression to your needs.

#### NOTE

Once customization has occurred, unchecking the **Customize** button will result in all customized changes being discarded in favor of the displayed **Key / Value** selections.

### Events

The trace events upon which this qualified expression is based.

#### NOTE

This field is labeled **Start Events** when defining a qualified state (see “Qualified States” on page 11-116).

### Browse

Presents a dialog allowing the user to select from a list of defined trace event names.

User-defined trace event names are associated with trace event ID numbers using an *event map file* (see “Event Map Files” on page 4-10).

#### NOTE

To select multiple items, hold the **Ctrl** key while selecting items in the list.

### Start Events

A set of trace events, any of which may mark the beginning of this qualified state (see “Qualified States” on page 11-116).

#### NOTE

In order for a trace event to be considered a *start event*, all other criteria specified in this qualified state must be met: **Start Condition, Processes, Threads, Nodes, and CPUs.**

### Browse

Presents a dialog allowing the user to select from a list of defined trace event names.

User-defined trace event names are associated with trace event ID numbers using an *event map file* (see “Event Map Files” on page 4-10).

#### NOTE

To select multiple items, hold the Ctrl key while selecting items in the list.

### End Events

A set of trace events, any of which may define the end of this qualified state (see “Qualified States” on page 11-116).

#### NOTE

In order for a trace event to be considered an *end event*, all other criteria specified in this qualified state must be met: **End Condition, Processes, Threads, Nodes, and CPUs.**

### Browse

Presents a dialog allowing the user to select from a list of defined trace event names.

User-defined trace event names are associated with trace event ID numbers using an *event map file* (see “Event Map Files” on page 4-10).

#### NOTE

To select multiple items, hold the Ctrl key while selecting items in the list.

## Condition

A boolean expression specifying criteria that must be met as part of this qualified event. (See “Expressions” on page 11-1 for more information.)

### NOTE

This field is labeled **Start Condition** when defining a qualified state (see “Qualified States” on page 11-116).

This field is labeled **Expansion** when defining a macro (see “Macros” on page 11-111).

## Start Condition

A boolean expression which specifies criteria that must be met for this qualified state to begin. (See “Expressions” on page 11-1 for more information.)

### NOTE

Currently, NightTrace does not supported nesting of states. Thus, once the conditions which satisfy a *start event* are met, no other instances of that state can begin until the **End Condition** has been met.

## Expansion

The expression to be substituted by the macro (see “Macros” on page 11-111).

## End Condition

A boolean expression which specifies criteria that must be met for this qualified state to complete. (See “Expressions” on page 11-1 for more information.)

## Processes

Specify the processes to which this qualified expression is restricted.

You may specify either PID values or the names of processes to which NightTrace has corresponding PID associations (e.g. from kernel trace data).

## Browse

Presents a dialog allowing the user to select from a list of the names of all processes that NightTrace discovers in the current trace data.

**NOTE**

To select multiple items, hold the Ctrl key while selecting items in the list.

**Threads**

Specify the threads to which this qualified expression is restricted.

You may specify either TID values or the names of threads to which NightTrace has corresponding TID associations (e.g. from kernel trace data).

**Browse**

Presents a dialog allowing the user to select from a list of the names of all threads that NightTrace discovers in the current trace data.

**NOTE**

To select multiple items, hold the Ctrl key while selecting items in the list.

**Nodes**

Specify the system node names to which this qualified expression is restricted.

**NOTE**

The **Nodes** field is only meaningful for datasets captured from more than one system and is used to differentiate between them. Use of the *RCIM* timing source on daemon invocations is required for time synchronization in such cases. See the *Real-Time Clock and Interrupt Module User's Guide* (0891082) for more information.

**Browse**

Presents a dialog allowing the user to select from a list of the names of all nodes that NightTrace discovers in the current trace data.

**NOTE**

Press the Ctrl key while selecting items in the list to select multiple items.

### **CPUs**

Specify which CPUs to which this qualified expression is restricted.

### **All CPUs**

All CPUs are selected when this checkbox is checked.

### **Name**

The name by which this qualified expression will be referenced. The user may either specify a name or use the name generated by NightTrace.

The following buttons appear at the bottom of the **Edit NightTrace Qualified Expression** dialog and have the specified meaning:

### **OK**

Saves all changes and closes the **Edit NightTrace Qualified Expression** dialog. New qualified expressions are added to the list in the **NightTrace Qualified Expressions** dialog (see “NightTrace Qualified Expressions” on page 11-119).

### **Revert**

Reverts all fields in the **Edit NightTrace Qualified Expression** dialog back to the values they had before any changes were made in this dialog.

### **Cancel**

Closes the **Edit NightTrace Qualified Expression** dialog without saving any changes.

### **Help**

Opens the HyperHelp viewer to the online help topic for this dialog.





## Search and Summarize



Searching for Points of Interest . . . . .	12-1
Search Options . . . . .	12-10
Summarizing Statistical Information . . . . .	12-12
Criteria . . . . .	12-14
Options . . . . .	12-26



## Search and Summarize

NightTrace makes it easier for you to pinpoint important trace events and numerically analyze aspects of your trace session.

“Searching for Points of Interest” on page 12-1 describes the Search NightTrace Events dialog and its usage.

“Summarizing Statistical Information” on page 12-12 describes the Summarize NightTrace Events dialog and its usage.

### Searching for Points of Interest

The Search NightTrace Events dialog allows you to locate areas of interest in your trace event file(s). This dialog allows you to provide search specifications and define conditions you wish to find in your trace event file(s).

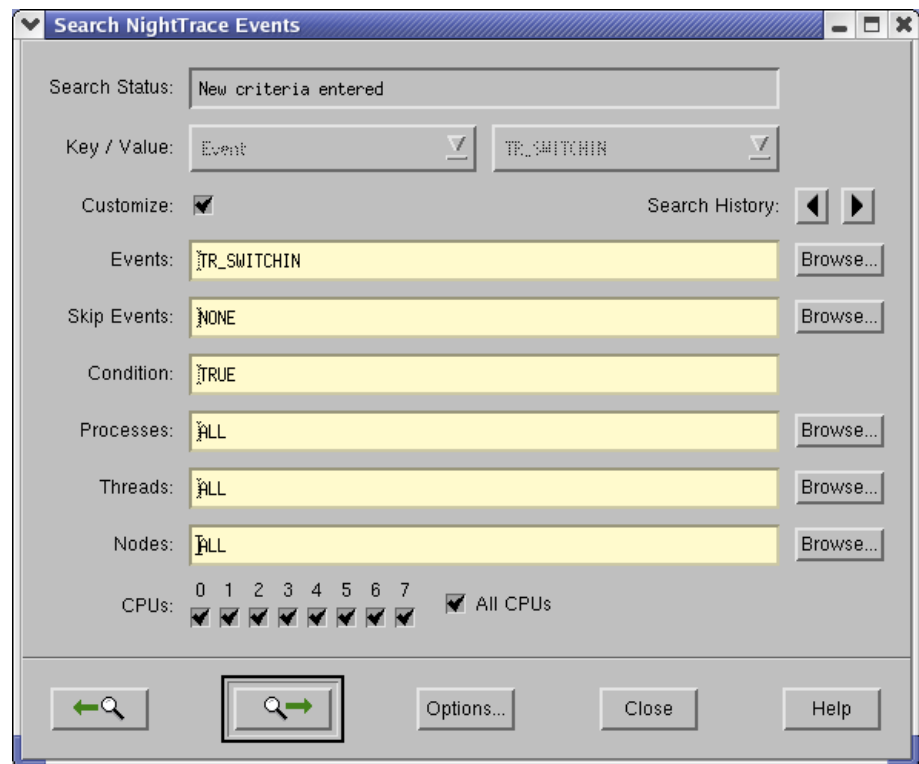


Figure 12-1. Search NightTrace Events dialog

The **Search NightTrace Events** dialog is opened by selecting the **Change Search Criteria...** item from the **Actions** menu on any display page (see “Change Search Criteria...” on page 9-21) or by pressing **Ctrl-F** from any display page.

#### **NOTE**

Once search criteria has been specified, subsequent searches for the same criteria can be made without opening the **Search NightTrace Events** dialog. Pressing the **>** or **<** keys executes a forward or backward search, respectively, using the current search criteria. (Note that it is not necessary to press the **Shift** key when using these accelerators.)

By default, in a new NightTrace session, if no search criteria has been entered, NightTrace will search for all events.

Search criteria is saved as part of the session configuration file (see “Session Configuration Files” on page 4-24). Specifying a session configuration file on subsequent invocations of NightTrace reloads all search criteria from that session designating the last search executed from that session as the current search criteria for the current session. (See “Invoking NightTrace” on page 4-1 and “Command-line Arguments” on page 4-9 for more information.)

#### **Search Status**

Displays the results from the current search.

When viewing criteria from previously-executed searches, the text:

    Cached search *n*

appears in this field where *n* is the number assigned to this set of search criteria in the search history (see “Search History” on page 12-5).

#### **Key / Value**

These two drop-down menus are used in combination to provide a quick and efficient method for specifying search criteria.

The value chosen for the **Key** will determine the type of information for which to search.

The **Value** drop-down menu provides a list of possible choices associated with the selected **Key**. The choices in this list are based on the trace dataset; for instance, if **Process ID** is selected for the **Key**, the **Value** drop-down menu will consist of those processes in the current dataset that logged trace events and/or those that were executing when kernel trace events were collected.

**NOTE**

To specify user trace events that do not have trace event names associated with them and therefore do not appear in the **Value** drop-drop menu, check the **Customize** checkbox and enter the numeric *trace event ID* in the appropriate field (e.g. **Events**, **Skip Events**, etc.).

The search criteria can be further customized by selecting the **Customize** checkbox and modifying the remaining fields.

The **Key** drop-down contains the following items:

Event

search for the event selected in the **Value** drop-down

Exclude Event

search for all events except the event selected in the **Value** drop-down

Tagged Event

search for the tagged event selected in the **Value** drop-down (see “Tag” on page 9-32)

Qualified Event

search for the qualified event (see “Qualified Events” on page 11-113) selected in the **Value** drop-down

Qualified State

search for the qualified state (see “Qualified States” on page 11-116) selected in the **Value** drop-down

Process ID

search for events associated with the process selected in the **Value** drop-down

Thread ID

search for events associated with the thread selected in the **Value** drop-down

System Call

search for those instances when the kernel starts or resumes execution of the particular system call selected in the **Value** drop-down

System Call Leave

search for those instances when the kernel exits or suspends execution of the particular system call selected in the **Value** drop-down

System Call Events

search for those instances when the kernel starts, resumes, exits, or suspends execution of the particular system call selected in the **Value** drop-down

Interrupt

search for those instances when the kernel starts executing the particular interrupt selected in the **Value** drop-down

Interrupt Leave

search for those instances when the kernel exits the particular interrupt selected in the **Value** drop-down

Interrupt Events

search for those instances when the kernel starts executing the particular interrupt selected in the **Value** drop-down as well as when the kernel exits that particular interrupt

Exception

search for those instances when the kernel starts or resumes execution of the particular exception selected in the **Value** drop-down

Exception Leave

search for those instances when the kernel exits or suspends execution of the particular exception selected in the **Value** drop-down

Exception Events

search for those instances when the kernel starts, resumes, exits, or suspends execution of the particular exception selected in the **Value** drop-down

## Customize

Use this option to further configure your search criteria.

When **Customize** is checked, the **Key / Value** drop-down menus become disabled (desensitized). The remaining fields (which are populated with values corresponding to the **Key / Value** pair) may be modified to tailor the search criteria to your needs.

### NOTE

Once customization has occurred, unchecking the **Customize** button will result in all customized changes being discarded in favor of the displayed **Key / Value** selections.

## Search History

These buttons allow the user to cycle through criteria from previously-executed searches.



Cycles backward to the previous set of criteria saved in the search history and displays those settings in the fields of this dialog.

If the first set of criteria saved in the cache is displayed, pressing this button will display the last set of search criteria saved in the cache.



Cycles forward to the next set of criteria saved in the search history and displays those settings in the fields of this dialog.

If the criteria from the most recently executed search is displayed, pressing this button will display the first set of search criteria saved in the cache.

---

## Events

Specify the trace events of interest for this particular search.

### Browse

Presents a dialog allowing the user to select from a list of defined trace event names.

User-defined trace event names are associated with trace event ID numbers using an *event map file* (see “Event Map Files” on page 4-10).

### NOTE

To select multiple items, hold the Ctrl key while selecting items in the list.

## Skip Events

Specify the trace events to be ignored for this particular search.

### Browse

Presents a dialog allowing the user to select from a list of defined trace event names.

User-defined trace event names are associated with trace event ID numbers using an *event map file* (see “Event Map Files” on page 4-10).

**NOTE**

To select multiple items, hold the Ctrl key while selecting items in the list.

**Condition**

A boolean expression specifying additional criteria for this particular search. (See “Expressions” on page 11-1 for more information.)

**Processes**

Specify the processes to which this search is restricted.

You may specify either PID values or the names of processes to which NightTrace has corresponding PID associations (e.g. from kernel trace data).

**Browse**

Presents a dialog allowing the user to select from a list of the names of all processes that NightTrace discovers in the current trace data.

**NOTE**

To select multiple items, hold the Ctrl key while selecting items in the list.

**Threads**

Specify the threads to which this search is restricted.

You may specify either TID values or the names of threads to which NightTrace has corresponding TID associations (e.g. from kernel trace data).

**Browse**

Presents a dialog allowing the user to select from a list of the names of all threads that NightTrace discovers in the current trace data.

**NOTE**

To select multiple items, hold the Ctrl key while selecting items in the list.



## Nodes

Specify the system node names to which this search is restricted.

### NOTE

The **Nodes** field is only meaningful for datasets captured from more than one system and is used to differentiate between them. Use of the *RCIM* timing source on daemon invocations is required for time synchronization in such cases. See the *Real-Time Clock and Interrupt Module User's Guide* (0891082) for more information.

## Browse

Presents a dialog allowing the user to select from a list of the names of all nodes that NightTrace discovers in the current trace data.

### NOTE

Press the **Ctrl** key while selecting items in the list to select multiple items.

## CPUs

Specify which CPUs to which this search is restricted.

### All CPUs

All CPUs are selected when this checkbox is checked.

The following buttons appear at the bottom of the **Search NightTrace Events** dialog and have the specified meaning:



Accelerator: <

**NOTE**

It is not necessary to press the **Shift** key when using this accelerator.

Furthermore, it is not necessary to have the **Search NightTrace Events** window open when using this accelerator (see "Searching for Points of Interest" on page 12-1). The search criteria specified from the previous search is used.

Searches backward from the current time for the state or event meeting the specified criteria according to the selected search options (see "Search Options" on page 12-10).

Results from this search appear in the message display area of the display page from which this search was executed (see "Message Display Area" on page 9-28).



Accelerator: >

**NOTE**

It is not necessary to press the **Shift** key when using this accelerator.

Furthermore, it is not necessary to have the **Search NightTrace Events** window open when using this accelerator (see "Searching for Points of Interest" on page 12-1). The search criteria specified from the previous search is used.

Searches forward from the current time for the state or event meeting the specified criteria according to the selected search options (see "Search Options" on page 12-10).

Results from this search appear in the message display area of the display page from which this search was executed (see "Message Display Area" on page 9-28).

**Options...**

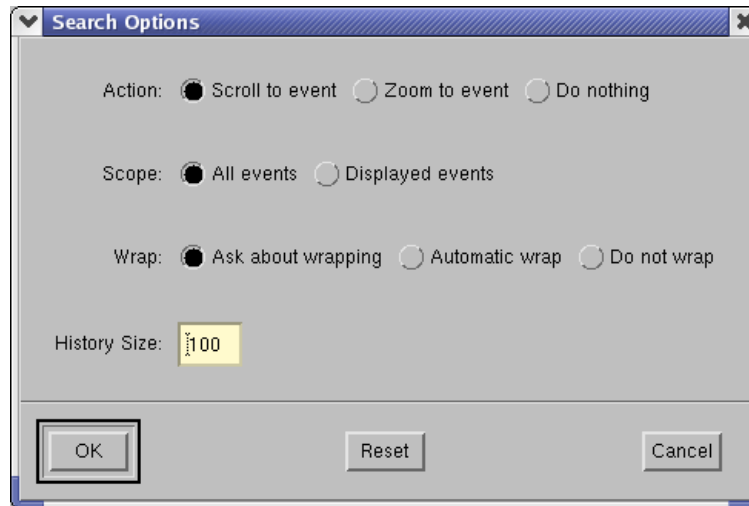
Opens the **Search Options** dialog (see "Search Options" on page 12-10) allowing the user to choose what action takes place when a trace event meets the specified search criteria, the scope of the search, wrapping preferences, and the number of items to be held in the search history.

**Close**

Closes the Search NightTrace Events dialog.

## Search Options

The **Search Options** dialog allows the user to choose what action takes place when a trace event meets the specified search criteria, the scope of the search, wrapping preferences, and the number of items to be held in the search history.



**Figure 12-2. Search Options dialog**

### Action

These radio buttons allow you to choose the action NightTrace takes if a trace event meets the specified criteria.

#### Scroll to event

NightTrace sets the current time to the time when the trace event occurred and moves the interval.

#### Zoom to event

Zoom out the interval end time (for forward searches) or the interval start time (for backward searches) to include the found trace event. The current time is updated accordingly.

#### Do nothing

NightTrace writes a message to the message display area of the display page without repositioning you on the grid or in the interval control area.

A side-effect of this setting is that repeatedly clicking on the **Search** push button does not find trace events after the first one found. This is because the current time has not changed.

## Scope

This setting determines the portion of the dataset to be included in the search.

### All events

All items in the dataset are included in the search.

### Displayed events

Only those items in the current *interval* are included in the search.

The current interval is defined to be the region delimited by the **Start Time** and **End Time** fields of the *interval control area* (see “Interval Control Area” on page 9-37).

## Wrap

This setting determines the behavior when the end of the dataset is reached during a search.

### Ask about wrapping

A dialog is presented to the user when the end of the dataset is reached, asking if the user would like to continue the search from the beginning of the dataset.

### Automatic wrap

When the end of the dataset is reached, the search is automatically continued from the beginning of the dataset.

### Do not wrap

The search does not continue when the end of the dataset is reached.

## History Size

The maximum number of searches to be saved in the search history cache (see “Search History” on page 12-5).

## Summarizing Statistical Information

The Summarize NightTrace Events dialog lets you get statistical information about trace events and states, allowing you to constrain the information to be summarized to your specifications. In addition, this dialog allows you to reposition the current time line to the state with either the shortest or longest duration as well as display a Data Graph (see “Data Graph” on page 10-8) showing the durations of each state on which the summary is based.

The Summarize NightTrace Events dialog is opened by selecting the Change Summary Criteria... item from the Actions menu on any display page (see “Change Summary Criteria...” on page 9-22) or by pressing Ctrl-Z from any display page.

The following checkbox appears at the top of the Summarize NightTrace Events dialog:

### Show summary results in this dialog

When this checkbox is checked, results from the summary appear at the top of the Summarize NightTrace Events dialog as well as in the message display area of the display page from which this summary was executed.

### NOTE

Summary results always appear in the message display area (see “Message Display Area” on page 9-28) regardless of this setting.

Figure 12-3 shows an example of summary results displayed in the Summarize NightTrace Events dialog.

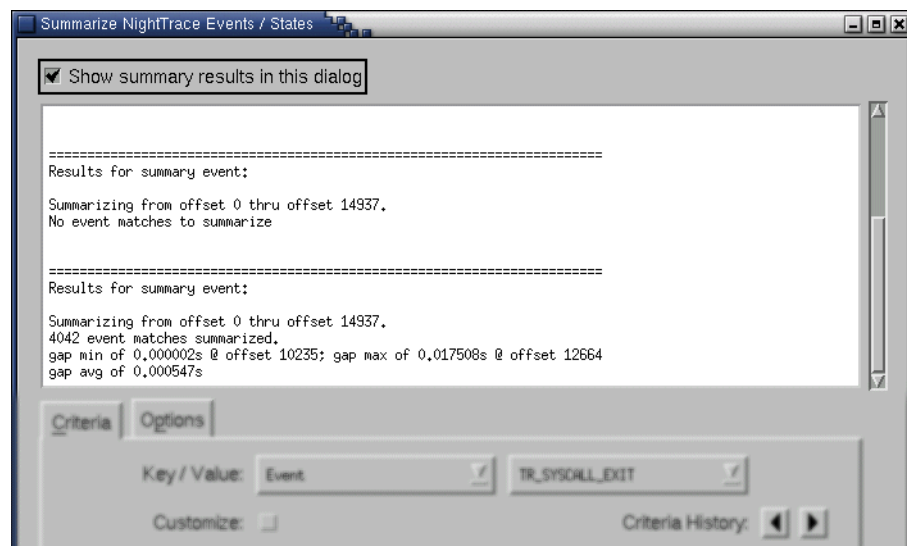


Figure 12-3. Summary results displayed in Summarize dialog

The **Summarize NightTrace Events** dialog is divided into a number of pages that contain specific information about the current summary. These pages are:

- **Criteria**

See “Criteria” on page 12-14 for more detailed information.

- **Options**

See “Options” on page 12-26 for more detailed information.

The following buttons appear at the bottom of the **Summarize NightTrace Events** dialog and have the specified meaning:

### **Summarize**

Performs a summary based on the criteria specified in this dialog according to the options specified on the **Options** page (see “Options” on page 12-26).

Results from this summary appear in the message display area of the display page from which this search was executed (see “Message Display Area” on page 9-28).

In addition, the user may check the **Show summary results in this dialog** checkbox (see “Show summary results in this dialog” on page 12-12) to view the results from the summary directly in the **Summarize NightTrace Events** dialog.

### **Clear Results**

Clears the cache containing the results of summaries performed in the current session.

### **Save Results...**

Presents a file selection dialog allowing the user to save the cache containing the results of the summaries to an external file.

### **Close**

Closes the **Summarize NightTrace Events** dialog.

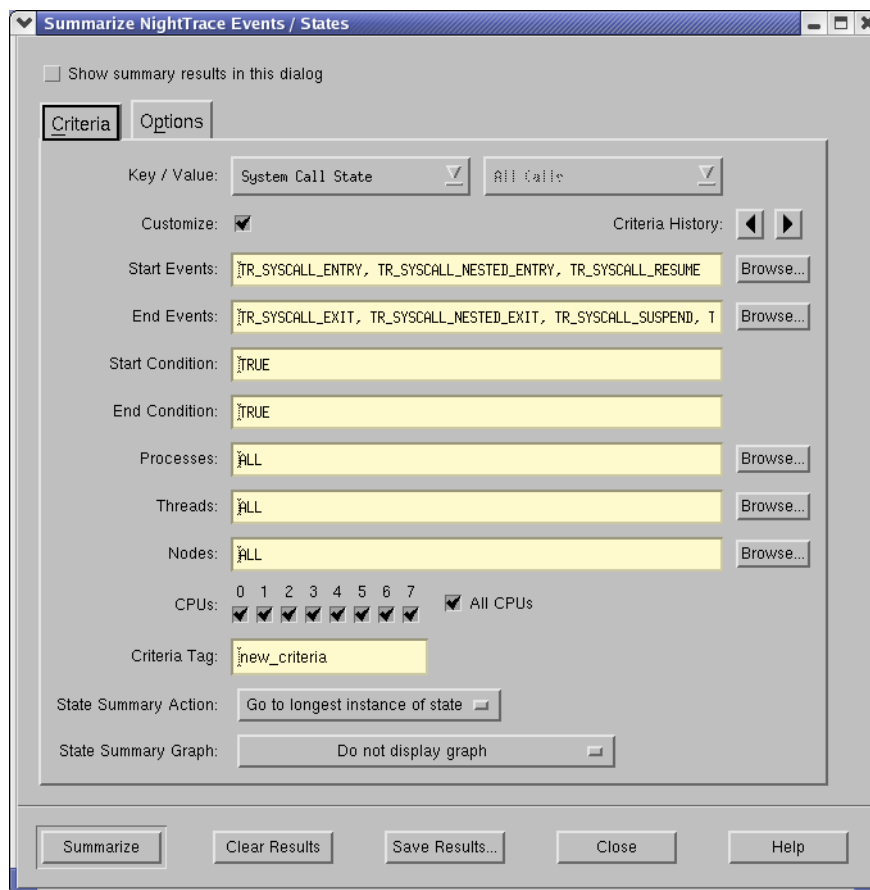
### **Help**

Opens the HyperHelp viewer to the online help topic for this dialog.

## Criteria

The **Criteria** page allows the user to specify the conditions on which a particular summary is based. Events of interest, starting and ending events for states of interest, and any other conditions including processes, threads, nodes, and CPUs to which this summary is restricted are all specified on this page.

In addition, when summarizing instances of a particular state, an option is provided to reposition the current timeline to either the shortest or longest instance of that state. The user may also request a state summary graph showing the durations of each state on which the summary is based.



**Figure 12-4. Summarize NightTrace Events dialog - Criteria page**

### Key / Value

These two drop-down menus are used in combination to provide a quick and efficient method for specifying summary criteria.

The value chosen for the **Key** will determine the type of information to be summarized.



The **Value** drop-down menu provides a list of possible choices associated with the selected **Key**. The choices in this list are based on the trace dataset; for instance, if **Process ID** is selected for the **Key**, the **Value** drop-down menu will consist of those processes in the current dataset that logged trace events and/or those that were executing when kernel trace events were collected.

### NOTE

To specify user trace events that do not have trace event names associated with them and therefore do not appear in the **Value** drop-down menu, check the **Customize** checkbox and enter the numeric *trace event ID* in the appropriate field (e.g. **Events**, **Start Events**, etc.). User-defined trace event names are associated with trace event ID numbers using an *event map file* (see “Event Map Files” on page 4-10).

The summary criteria can be further customized by selecting the **Customize** checkbox and modifying the remaining fields.

The **Key** drop-down contains the following items:

Event

summarize the number of occurrences of the event selected in the **Value** drop-down

Exclude Event

summarize the number of occurrences of all events except the one selected in the **Value** drop-down

Qualified Event

summarize all occurrences of the qualified event (see “Qualified Events” on page 11-113) selected in the **Value** drop-down

Qualified State

summarize all occurrences of the qualified state (see “Qualified States” on page 11-116) selected in the **Value** drop-down

Process ID

summarize all events associated with the process selected in the **Value** drop-down

Thread ID

summarize all events associated with the thread selected in the **Value** drop-down

System Call

summarize those instances when the kernel starts or resumes execution of the particular system call selected in the **Value** drop-down

System Call Leave

summarize those instances when the kernel exits or suspends execution of the particular system call selected in the **Value** drop-down

System Call Events

summarize those instances when the kernel starts, resumes, exits, or suspends execution of the particular system call selected in the **Value** drop-down

Interrupt

summarize those instances when the kernel starts executing the particular interrupt selected in the **Value** drop-down

Interrupt Leave

summarize those instances when the kernel finishes executing the particular interrupt selected in the **Value** drop-down

Interrupt Events

summarize those instances when the kernel starts executing the particular interrupt selected in the **Value** drop-down as well as when the kernel exits that particular interrupt

Exception

summarize those instances when the kernel starts or resumes execution of the particular exception selected in the **Value** drop-down

Exception Leave

summarize those instances when the kernel exits or suspends execution of the particular exception selected in the **Value** drop-down

Exception Events

summarize those instances when the kernel starts, resumes, exits, or suspends execution of the particular exception selected in the **Value** drop-down

## Customize

Use this option to further configure your summary criteria.

When **Customize** is checked, the **Key / Value** drop-down menus become disabled (desensitized). The remaining fields (which are populated with values corre-

sponding to the **Key / Value** pair) may be modified to tailor the qualified expression to your needs.

### NOTE

Once customization has occurred, unchecking the **Customize** button will result in all customized changes being discarded in favor of the displayed **Key / Value** selections.

### Criteria History

These buttons allow the user to cycle through criteria from previously-executed summaries.



Cycles backward to the previous set of criteria saved in the summary history and displays those settings in the fields of this dialog.

If the first set of criteria saved in the cache is displayed, pressing this button will display the last set of summary criteria saved in the cache.



Cycles forward to the next set of criteria saved in the summary history and displays those settings in the fields of this dialog.

If the criteria from the most recently executed summary is displayed, pressing this button will display the first set of summary criteria saved in the cache.

---

### Events

The trace events upon which this summary is based.

### NOTE

This field is labeled **Start Events** when summarizing the instances of states (see “Qualified States” on page 11-116).

### Browse

Presents a dialog allowing the user to select from a list of defined trace event names.

User-defined trace event names are associated with trace event ID numbers using an *event map file* (see “Event Map Files” on page 4-10).

**NOTE**

To select multiple items, hold the Ctrl key while selecting items in the list.

**Start Events**

The events that can be considered as the beginning of a state to be included in this summary.

**NOTE**

In order for a trace event to be considered a *start event*, all other criteria specified for this summary must be met: **Start Condition, Processes, Threads, Nodes, and CPUs.**

**Browse**

Presents a dialog allowing the user to select from a list of defined trace event names.

User-defined trace event names are associated with trace event ID numbers using an *event map file* (see “Event Map Files” on page 4-10).

**NOTE**

To select multiple items, hold the Ctrl key while selecting items in the list.

**End Events**

The events that can be considered as the end of a state to be included in this summary.

**NOTE**

In order for a trace event to be considered an *end event*, all other criteria specified for this summary must be met: **Start Condition, Processes, Threads, Nodes, and CPUs.**

**Browse**

Presents a dialog allowing the user to select from a list of defined trace event names.

User-defined trace event names are associated with trace event ID numbers using an *event map file* (see “Event Map Files” on page 4-10).

**NOTE**

To select multiple items, hold the Ctrl key while selecting items in the list.

**Condition**

A boolean expression specifying criteria that must be met when summarizing the instances of events. (See “Expressions” on page 11-1 for more information.)

**NOTE**

This field is labeled **Start Condition** when summarizing the instances of states (see “Qualified States” on page 11-116).

**Start Condition**

A boolean expression specifying criteria that must be met at the beginning of a particular state for it to be included in the summary. (See “Expressions” on page 11-1 for more information.)

**End Condition**

A boolean expression specifying criteria that must be met at the end of a particular state for it to be included in the summary. (See “Expressions” on page 11-1 for more information.)

**Processes**

Specify the processes to which this summary is restricted.

You may specify either PID values or the names of processes to which NightTrace has corresponding PID associations (e.g. from kernel trace data).

**Browse**

Presents a dialog allowing the user to select from a list of the names of all processes that NightTrace discovers in the current trace data.

**NOTE**

To select multiple items, hold the Ctrl key while selecting items in the list.

## Threads

Specify the threads to which this summary is restricted.

You may specify either TID values or the names of threads to which NightTrace has corresponding TID associations (e.g. from kernel trace data).

### Browse

Presents a dialog allowing the user to select from a list of the names of all threads that NightTrace discovers in the current trace data.

#### NOTE

To select multiple items, hold the Ctrl key while selecting items in the list.

## Nodes

Specify the system node names to which this summary is restricted.

#### NOTE

The **Nodes** field is only meaningful for datasets captured from more than one system and is used to differentiate between them. Use of the *RCIM* timing source on daemon invocations is required for time synchronization in such cases. See the *Real-Time Clock and Interrupt Module User's Guide* (0891082) for more information.

### Browse

Presents a dialog allowing the user to select from a list of the names of all nodes that NightTrace discovers in the current trace data.

#### NOTE

Press the Ctrl key while selecting items in the list to select multiple items.

## CPUs

Specify which CPUs to which this summary is restricted.

### All CPUs

All CPUs are selected when this checkbox is checked.

## Criteria Tag

The name by which this summary will be referenced. The user may either specify a name or use the system-generated default.

The **Criteria Tag** may be used with the **--summary** option when performing command-line summaries (see “Command-line Options” on page 4-1 and “Summary Criteria” on page 4-5).

If a state summary graph is requested (see “State Summary Graph” on page 12-21), a qualified state (see “Qualified States” on page 11-116) based on the specified criteria is created. This qualified state takes the name specified in the **Criteria Tag** field and is added to the list of qualified expressions for this session (see “NightTrace Qualified Expressions” on page 11-119). This qualified state can then be used to search for other instances by specifying its name in the **Search NightTrace Events** dialog (see “Searching for Points of Interest” on page 12-1).

### NOTE

The list of qualified expressions for a particular session can be found in the **NightTrace Qualified Expressions** dialog (see “NightTrace Qualified Expressions” on page 11-119).

## State Summary Action

The user may select to position the current timeline to either the shortest or longest instance of the state on which a particular summary is based or may elect not to reposition the timeline at all.

The user may select one of the following choices from the drop-down menu:

### Go to shortest instance of state

The current timeline is repositioned to the end of the state having the shortest duration.

### Go to longest instance of state

The current timeline is repositioned to the end of the state having the longest duration.

### Do not move timeline

The current timeline is not repositioned.

## State Summary Graph

Display a Data Graph (see “Data Graph” on page 10-8) showing either the durations of each state on which the summary is based or the gaps between the states.

## NOTE

The scale factor for these graphs is automatically determined by the shortest and longest values found. This can sometimes have the effect of obscuring useful data. Consider a situation where 99% of the state instances had a duration on the order of 10-30 microseconds, but a single instance lasted 500000 microseconds. The resulting graph would have a single large spike with the details of the remaining states difficult to ascertain. Use the ( $n \times$  Std. Dev.) menu items in such instances.

The user may select one of the following choices from the drop-down menu:

### Display graph of state durations

Display a Data Graph showing the durations of each state on which the summary is based.

### Display graph of state durations (1 x Std. Dev.)

Display a Data Graph showing the durations of each state on which the summary is based.

The scale factor for the graph is automatically determined by the minimum and maximum state duration that fall within one standard deviation of the actual minimum and maximum. All state durations will appear on the graph.

### Display graph of state durations (2 x Std. Dev.)

Display a Data Graph showing the durations of each state on which the summary is based.

The scale factor for the graph is automatically determined by the minimum and maximum state duration that fall within two standard deviations of the actual minimum and maximum. All state durations will appear on the graph.

### Display graph of state gaps

Display a Data Graph showing the durations of the gap between the states on which the summary is based.

### Display graph of state gaps (1 x Std. Dev.)

Display a Data Graph showing the durations of the gap between the states on which the summary is based.

The scale factor for the graph is automatically determined by the minimum and maximum duration of the gaps between states that fall within one standard deviation of the actual minimum and maximum. All state durations will appear on the graph.



### Display graph of state gaps (2 x Std. Dev.)

Display a Data Graph showing the durations of the gap between the states on which the summary is based.

The scale factor for the graph is automatically determined by the minimum and maximum duration of the gaps between states that fall within two standard deviations of the actual minimum and maximum. All state durations will appear on the graph.

### Do not display graph

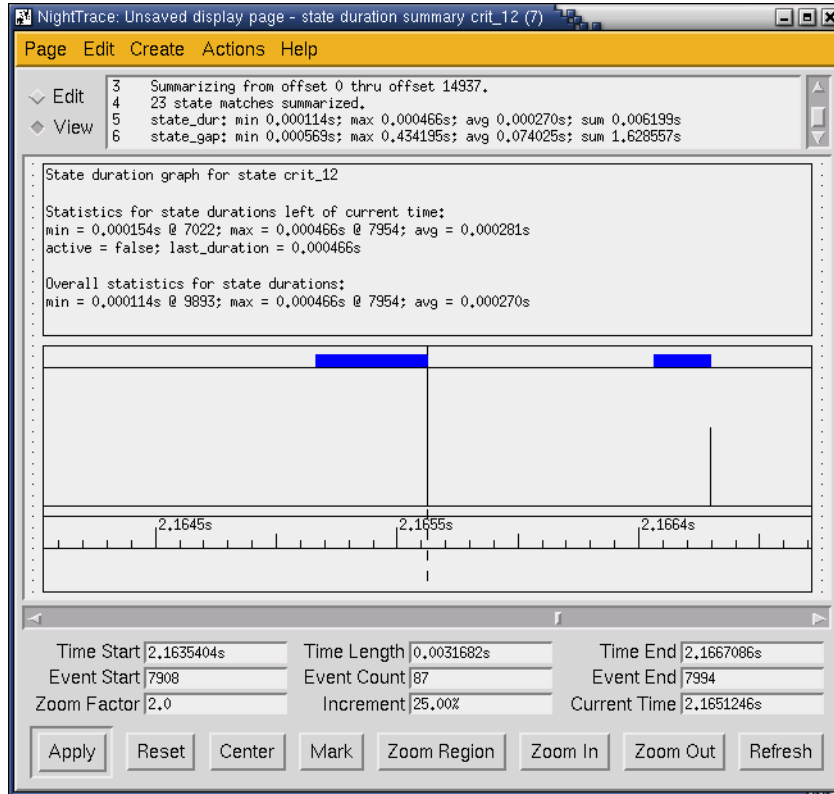
No state summary graph is displayed.

When a state summary graph is requested, a qualified state (see “Qualified States” on page 11-116) based on the specified criteria is created. This qualified state takes the name specified in the **Criteria Tag** field and is added to the list of qualified expressions for this session. This qualified state can then be used to search for other instances by specifying its name in the **Search NightTrace Events** dialog (see “Searching for Points of Interest” on page 12-1).

### **NOTE**

The list of qualified expressions for a particular session can be found in the **NightTrace Qualified Expressions** dialog (see “NightTrace Qualified Expressions” on page 11-119).

Figure 12-5 shows an example of a state summary graph.



**Figure 12-5. State Summary Graph**

In Figure 12-5:

- the blue bars represent instances of the state specified in the summary
- the vertical lines represent the duration of that instance of the state

**NOTE**

Since this is a Data Graph (see “Data Graph” on page 10-8), taller lines represent longer durations.

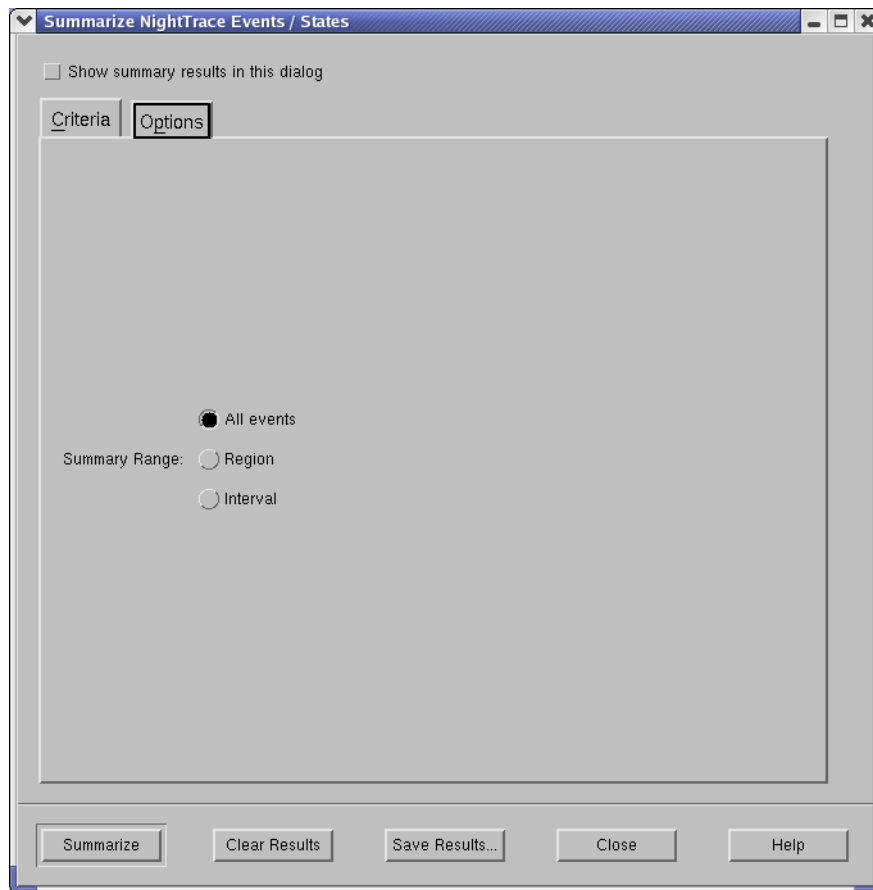
- the statistics above the graphs provide information about the *last completed instance of the state* and the overall statistics covering the time interval specified on the Options page of the Summarize NightTrace Events dialog (which defaults to the entire trace session - see “Options” on page 12-26)

**NOTE**

The statistics for the *last completed instance of the state* are updated as you change the position of the *current time line*.

## Options

The options on this page lets you limit the summary to the current interval, to the time between a mark and the current time, or to the entire trace session.



**Figure 12-6. Summarize NightTrace Events dialog - Options page**

### Summary Range

The user may select one of the following choices from the drop-down menu:

#### All events

Summarize data throughout the trace session.

#### Region

Summarize data only between the mark and the current time (see .

## Interval

Summarize data included in the current *interval* only.

The current interval is defined to be the region delimited by the **Start Time** and **End Time** fields of the *interval control area* (see “Interval Control Area” on page 9-37).



---

Default Kernel Trace Points . . . . .	13-1
Context Switch Trace Event . . . . .	13-2
Interrupt Trace Events . . . . .	13-2
Exception Trace Events . . . . .	13-3
Syscall Trace Events . . . . .	13-4
Kernel Trace Points Not Enabled By Default . . . . .	13-5
Page Fault Event . . . . .	13-5
Protection Fault Event . . . . .	13-5
Viewing Kernel Trace Event Files . . . . .	13-6
Kernel Display Pages . . . . .	13-6
Node and CPU Information . . . . .	13-7
Running Process Information . . . . .	13-8
Node Information . . . . .	13-8
Context Switch Information . . . . .	13-9
Interrupt Information . . . . .	13-9
Exception Information . . . . .	13-10
Syscall Information . . . . .	13-12
Color Information . . . . .	13-13
Kernel String Tables . . . . .	13-13
Kernel Reference . . . . .	13-15
Interrupts . . . . .	13-15
Non-Device-Related Interrupts . . . . .	13-16
Device-Related Interrupts . . . . .	13-16
Exceptions . . . . .	13-17
Syscalls . . . . .	13-18





## Tracing the Kernel

This chapter provides a description of the trace points logged by the kernel. It also discusses the steps required to produce a highly detailed picture of kernel activity with NightTrace. This lets you customize the default NightTrace kernel display pages or combine kernel information with user-application trace information.

Kernel trace event files are logged in raw format by the kernel trace daemon. `ntrace` accepts files of this type as arguments. When it detects such a file on the command line, or when the `-r` option indicates such a file should be displayed, it automatically filters the raw data file and creates two new files. The first file created is the filtered data, which contains trace events in a manner suitable for display within NightTrace. This file is saved with a pathname constructed from the original raw kernel trace event filename with a `“.ntf”` suffix appended to it. The second file saved is commonly referred to as the “vectors” file. It contains tables that are specific to the actual raw data. The “vectors” file is saved with a pathname constructed from the original raw kernel trace event filename with a `“.vec”` suffix appended to it. A more detailed description of the vectors file is given subsequently in this chapter.

On subsequent invocations of NightTrace, either the raw kernel file may be specified, or, alternatively, the `“.ntf”` and `“.vec”` files may be specified together.

### Default Kernel Trace Points

The following kernel trace points are enabled by default:

- `TR_SWITCHIN`
- `TR_INTERRUPT_ENTRY` and `TR_INTERRUPT_EXIT`
- `TR_EXCEPTION_ENTRY` and `TR_EXCEPTION_EXIT`
- `TR_SYSCALL_ENTRY`
- `TR_IO_VNODE`
- `TR_ALT_INT_DISPATCH`
- `TR_PROCESS_NAME`

These default kernel trace points are required to get meaningful kernel performance data in a KernelTrace trace event file. However, these trace points are not the only trace points that you will see with NightTrace when viewing kernel data. Specifically, the following trace points are introduced during raw kernel trace data processing by NightTrace:

- `TR_SYSCALL_EXIT`
- `TR_SYSCALL_SUSPEND` and `TR_SYSCALL_RESUME`

- TR\_EXCEPTION\_SUSPEND and TR\_EXCEPTION\_RESUME

The following sections discuss the trace events that you will see in NightTrace as a result of logging the default kernel trace points.

## Context Switch Trace Event

There is only one context switch trace event:

TR\_SWITCHIN *arg1*

This trace event is logged whenever a process has been switched in and is ready to be run on a specific CPU. Because only one process can run on a given CPU at a time, this trace event also signifies that the process that was running on the CPU immediately prior to the context switch trace event has been switched out and can no longer run. This trace event has one argument:

*arg1*            The numeric 32-bit global process identifier (PID) of the process being switched in. This information is redundant, since it is identical to the PID that is already associated with the trace event. A PID of 0 indicates that the CPU is idle.

The 32-bit global process identifier uniquely identifies the running process on the system. This identifier is identical to the return value of the `_lwp_global_self()` system call for PowerMAX OS and the `getpid()` system call under RedHawk Linux. See “pid()” on page 11-19.

## Interrupt Trace Events

There are two trace events associated with interrupts:

TR\_INTERRUPT\_ENTRY *arg1 arg2 arg3*

This trace event is logged whenever an interrupt is entered. It has three arguments:

*arg1*            The interrupt vector number that indicates the type of interrupt. This is an index into the `vector` string table that is contained within the `vectors` file generated by NightTrace when consuming kernel data. For more information about the `vector` string table, see “Kernel String Tables” on page 13-13.

*arg2*            The interrupt nesting level used by the pre-defined kernel pages to graph the different heights associated with the nesting level. This argument will be 1 for the first interrupt, 2 for a second interrupt that

interrupted the first interrupt, 3 for a third interrupt that interrupted the second interrupt, etc.

*arg3* The interrupt vector number of the previous interrupt that this interrupt entry is interrupting, if any.

TR\_INTERRUPT\_EXIT *arg1 arg2 arg3*

This trace event is logged whenever an interrupt is exited. Its arguments are identical to those of the TR\_INTERRUPT\_ENTRY trace event.

## Exception Trace Events

There are four trace events associated with exceptions:

TR\_EXCEPTION\_ENTRY *arg1*

This trace event is logged whenever an exception is entered. It has one argument:

*arg1* The exception vector number that indicates the type of exception. This is an index into the `vector` string table that is contained within the `vectors` file. For more information about the `vector` string table, see “Kernel String Tables” on page 13-13.

TR\_EXCEPTION\_SUSPEND *arg1*

This trace event is logged whenever an exception is suspended by a context switch. It has one argument that is identical to the argument logged with the TR\_EXCEPTION\_ENTRY trace event.

TR\_EXCEPTION\_RESUME *arg1*

This trace event is logged whenever an exception is resumed (i.e., the process that caused the exception to occur, which was switched out before the exception could be completed, is switched back in). A TR\_EXCEPTION\_RESUME trace event will always follow a TR\_EXCEPTION\_SUSPEND event, unless the process is being switched in for the first time since kernel tracing began.

It is possible for several TR\_EXCEPTION\_SUSPEND–TR\_EXCEPTION\_RESUME trace event pairs to occur if the process is switched in and out several times before the exception completes.

The TR\_EXCEPTION\_RESUME trace event has one argument that is identical to the argument logged with the TR\_EXCEPTION\_ENTRY trace event.

TR\_EXCEPTION\_EXIT *arg1*

This trace event is logged whenever an exception is completed. It has one argument that is identical to the argument that is logged with the TR\_EXCEPTION\_ENTRY trace event.

## Syscall Trace Events

There are four trace events associated with syscalls:

`TR_SYSCALL_ENTRY arg1 arg2 arg3`

This trace event is logged whenever a syscall is entered. It has three arguments:

- arg1* This argument is always zero for historical reasons.
- arg2* The syscall number that identifies the syscall. This is an index into the pre-defined `syscall` string table.
- arg3* The device number that indicates the type of device that is associated with the syscall, if any. This is an index into the pre-defined `device` string table.

For more information about the pre-defined `syscall` and `device` string tables, see “Kernel String Tables” on page 13-13.

`TR_SYSCALL_SUSPEND arg1 arg2 arg3`

This trace event is logged whenever a syscall is suspended by a context switch. It has three arguments that are identical to the arguments logged with the `TR_SYSCALL_ENTRY` trace event.

`TR_SYSCALL_RESUME arg1 arg2 arg3`

This trace event is logged whenever a syscall is resumed (i.e., the process that caused the syscall to occur, which was switched out before the syscall could be completed, is switched back in). A `TR_SYSCALL_RESUME` trace event will always follow a `TR_SYSCALL_SUSPEND` trace event, unless the process is being switched in for the first time since kernel tracing began.

It is possible for several `TR_SYSCALL_SUSPEND`–`TR_SYSCALL_RESUME` trace event pairs to occur if the process is switched in and out several times before the syscall completes.

The `TR_SYSCALL_RESUME` trace event has three arguments that are identical to the arguments logged with the `TR_SYSCALL_ENTRY` trace event. However, if a `TR_SYSCALL_RESUME` trace event does not follow a `TR_SYSCALL_SUSPEND` trace event (i.e., it is the first syscall trace event logged by the process since kernel tracing began) *arg2* identifies the syscall as “can't determine.”

`TR_SYSCALL_EXIT arg1 arg2 arg3`

This trace event is logged whenever a syscall is completed. It has three arguments that are identical to the arguments logged with the `TR_SYSCALL_ENTRY` trace event.

## Kernel Trace Points Not Enabled By Default

There are several kernel trace points which are not enabled by default but two of them deserve special mention. These two events allow you to determine areas in your application code where address faults are occurring, to minimize such faults, and thus improve the application's performance. The following sections discuss the page fault and protection fault kernel trace points.

### Page Fault Event

There is one page fault trace event:

```
TR_PAGEFLT_ADDR arg1 arg2 arg3
```

This trace event is logged whenever a kernel or user page fault occurs. The page fault can be either on a data address or on an instruction address. This trace event is not enabled by default because, depending upon system activity, page faults may occur reasonably frequently. This trace event has three arguments:

- |             |   |
|-------------|---|
| <i>arg1</i> | The data address which caused the page fault. If the page fault occurred on an instruction, this will be set to zero.   |
| <i>arg2</i> | The program counter value at the time of the page fault.  |
| <i>arg3</i> | The flag indicating whether the fault occurred on a kernel address or on a user address. A value of zero indicates that the fault occurred on a user address. A value of one indicates that the fault occurred on a kernel address. |

### Protection Fault Event

There is one protection fault trace event:

```
TR_PROTFLT_ADDR arg1 arg2 arg3
```

This trace event is logged whenever a kernel or user protection fault occurs. The protection fault can be either on a data address or on an instruction address. This trace event is not enabled by default because, depending upon system activity, protection faults may occur reasonably frequently. This trace event has three arguments:

- |             |   |
|-------------|---|
| <i>arg1</i> | The data address which caused the protection fault. If the protection fault occurred on an instruction, then this will be set to zero.        |
| <i>arg2</i> | The program counter value at the time of the protection fault.  |
| <i>arg3</i> | The flag indicating whether the fault occurred on a kernel address or on a user address. A value of zero indicates that the fault occurred on |

a user address. A value of one indicates that the fault occurred on a kernel address.

## Viewing Kernel Trace Event Files

NightTrace provides pre-defined kernel display pages to view kernel trace data (see “Kernel Display Pages” on page 13-6).

In addition, you may customize a kernel display page using the **Build Custom Kernel Page** dialog (see “Build Custom Kernel Page” on page 5-21) which is accessed by selecting the **Custom Kernel Page...** menu item from the **Pages** menu on the NightTrace Main Window (see “Custom Kernel Page...” on page 5-19).

## Kernel Display Pages

Figure 13-1 shows a sample kernel display page.

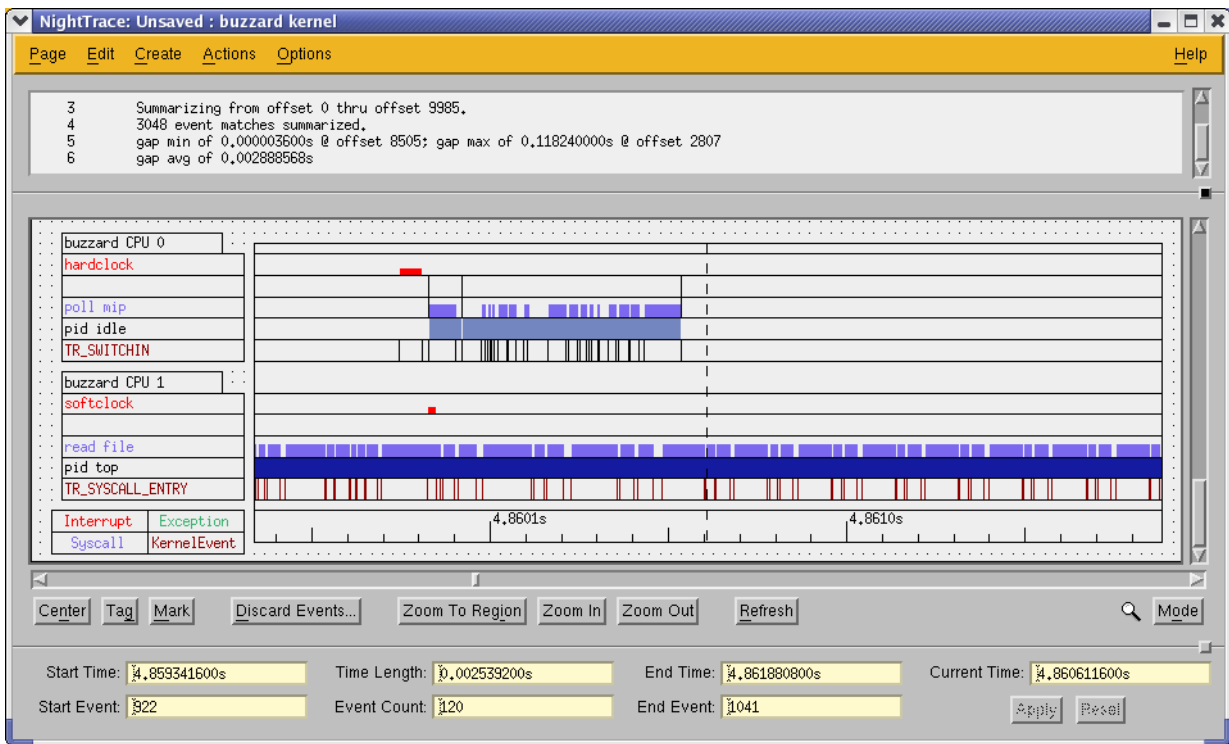
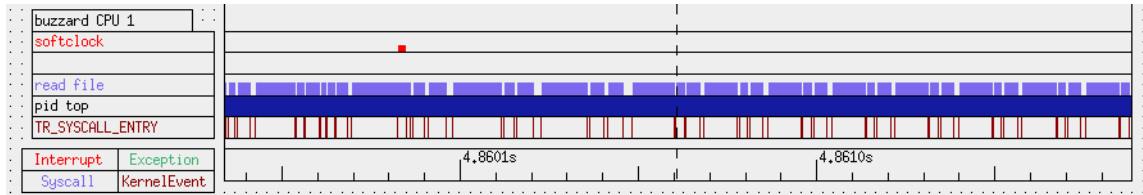


Figure 13-1. Sample Kernel Display Page



**Figure 13-2. Per-CPU Information**

There are several pieces of information being displayed for each CPU. The position of the current time line determines the values that appear on the kernel display pages. Moving the current time line within the current interval does not change the graphical displays. However, the textual displays always reflect the last values prior to the current time line.

The following sections discuss all of the different pieces of information in detail.

## Node and CPU Information

Figure 13-3 shows the Grid Label (see “Grid Label” on page 10-4) that appears on kernel display pages which displays information about the node and CPU corresponding to the trace data being displayed.

buzzard CPU 1

**Figure 13-3. Node and CPU Box**

The node identifies the node from which the displayed data was obtained.

The CPU identifies the logical CPU to which the displayed data corresponds. Logical CPU numbers are related to, but not necessarily identical to, physical CPU numbers.

Each CPU in a system has a four-bit physical CPU number. The physical CPU number is dependent on which card slot the CPU card containing the CPU is in and which location on the card the CPU is in. The low two bits of the number specify the location on the card that the CPU is in. These bits are either 00 for the first CPU location or 01 for the second. The high two bits of the physical CPU number contain the CPU card slot number. These bits can be 00, 01, 10, or 11 (or, in decimal, 0, 1, 2, or 3).

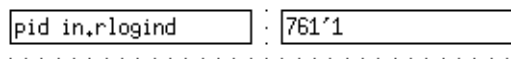
For simplicity, most kernel utilities translate the physical CPU numbers into logical CPU numbers. The mapping is accomplished by listing the physical CPU numbers of all configured CPUs in ascending order and then numbering them sequentially, starting with zero. For example, a four-CPU system having two CPUs on a card in slot 1 and two CPUs

on a card in slot 3 will have physical CPU numbers 4 (0100), 5 (0101), 12 (1100) and 13 (1101). Table 13-1 shows the logical CPU mapping of this example system.

**Table 13-1. Example Logical CPU Mapping**

Physical CPU Number	Logical CPU Number
4 (0100)	0
5 (0101)	1
12 (1100)	2
13 (1101)	3

## Running Process Information



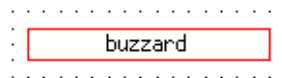
**Figure 13-4. Running Process Boxes**

Figure 13-4 shows two examples of running process boxes. The running process box shows the process that is executing at the current time on the associated CPU. The process is listed by name, or by its raw PID and LWPID if no name is available. See “Processes” on page 10-51 for more information about PIDs, raw PIDs and LWPIDs.

You can supply NightTrace trace event files to **ntrace** along with converted KernelTrace trace event files. NightTrace uses the process names of all processes that logged trace events when displaying the running process.

The running process box is a Data Box (“Data Box” on page 10-5). See “Configuring Display Objects” on page 10-15 for more information on configuring Data Boxes.

## Node Information



**Figure 13-5. Node Box**

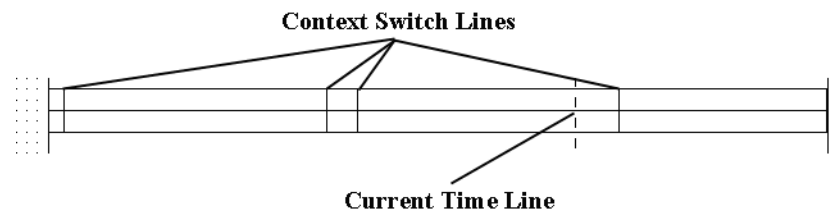
Figure 13-4 shows a node box. The node box simply identifies which node the displayed data corresponds to.



**NOTE**

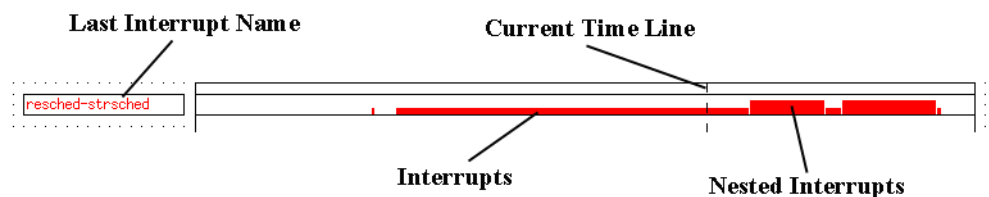
The node information is displayed only when NightTrace is configured to use an RCIM to timestamp events.

The node box is a Grid Label (“Grid Label” on page 10-4). See “Configuring Display Objects” on page 10-15 for more information on configuring Grid Labels.

**Context Switch Information**

**Figure 13-6. Context Switch Lines**

Figure 13-6 shows an example of several context switch lines. *Context switch lines* are superimposed on the exception and syscall graphs. They indicate that the kernel has switched out the process that was previously running on the CPU and switched in a new process. There is a direct correlation between context switch lines and the running process box: the running process box shows the process associated with the context switch line that immediately precedes the current time line.

**Interrupt Information**

**Figure 13-7. Last Interrupt Box and Interrupt Graph**

Figure 13-7 shows a last interrupt box and an interrupt graph. The interrupt graph displays a state that is drawn whenever an interrupt is executing on the associated CPU. Interrupts can be interrupted while executing, and the interrupt graph shows this interrupt nesting by increasing the height of the state bar. Although interrupts can nest, all interrupts must complete before the process they interrupt can be switched out. Therefore, you will never see a context switch occur in the middle of an interrupt.

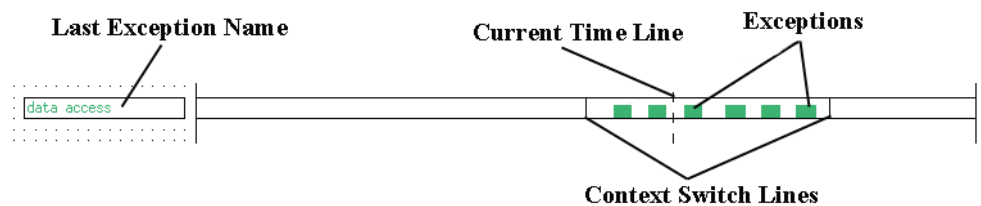
The last interrupt box displays the name of the last interrupt prior to the current time line that executed (and may still be executing) on the associated CPU. It can be used with the interrupt graph to identify any interrupts that are currently visible on the graph. Simply move the current time line onto a graphed interrupt, and the last interrupt box will update to display the name of the interrupt.

Because the last interrupt box displays the name of the last interrupt that executed, it is possible for there to be no interrupts visible on the interrupt graph even though the last interrupt box contains a valid interrupt name. This just signifies that the last interrupt on the CPU ended prior to the beginning of the current interval.

An interrupt that is seen very often is the hardclock interrupt, which usually accounts for 15% of the total number of trace events logged by the kernel. If you are not interested in hardclock interrupts, they can be ignored by NightTrace, improving performance and readability. See “Command-line Options” on page 4-1 for more information.

The last interrupt box is a Data Box (“Data Box” on page 10-5) and the last interrupt graph is a Data Graph (“Data Graph” on page 10-8). See “Configuring Display Objects” on page 10-15 for more information on configuring Data Boxes and Data Graphs.

## Exception Information



**Figure 13-8. Last Exception Box and Exception Graph**

Figure 13-8 shows a last exception box and an exception graph. The exception graph displays a state that is drawn whenever an exception is executing on the associated CPU. Unlike interrupts, exceptions cannot nest, so they are always graphed with the same height.

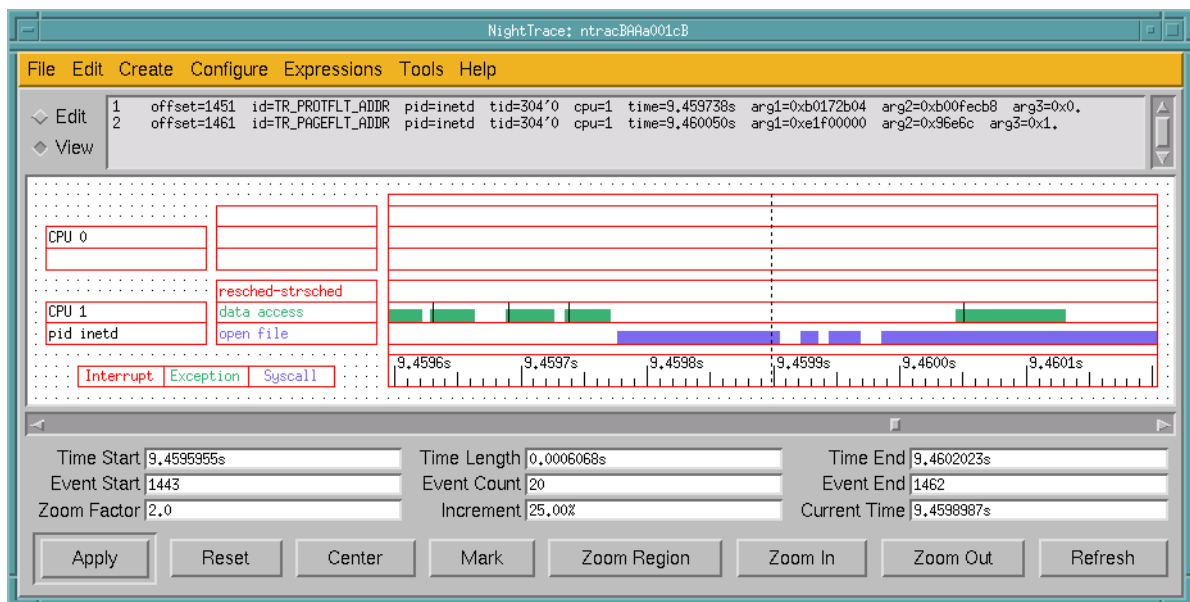
Context switch lines are superimposed on exception graphs. It is common to see a context switch line at what looks like the very end (or beginning) of an exception. Usually, this does not indicate that the exception has ended, only that it has been suspended because the process that originated the exception has switched out. The exception resumes when the process is switched back in again. An example of an exception being suspended and resumed can be seen at the left end of the exception graph in Figure 13-8.

The last exception box displays the last exception prior to the current time line that executed (and may still be executing) on the associated CPU. It can be used with the exception graph to identify any exceptions that are currently visible on the graph. Simply move the current time line onto a graphed exception, and the last exception box will update to display the name of the exception.

Because the last exception box displays the name of the last exception that executed, it is possible for there to be no exceptions visible on the exception graph even though the last exception box contains a valid exception name. This just signifies that the last exception on the CPU ended prior to the beginning of the current interval.

The last exception box is a Data Box (“Data Box” on page 10-5) and the last exception graph is a State Graph (see “State Graph” on page 10-7). See “Configuring Display Objects” on page 10-15 for more information on creating and configuring Data Boxes and State Graphs.

Lines indicating TR\_PAGEFLT\_ADDR and TR\_PROTFLT\_ADDR events are also superimposed on exception graphs. Exception graphs display these trace points to allow you to obtain a formatted dump of them in the message display area by clicking on the events with mouse button 2. An example of a TR\_PAGEFLT\_ADDR and a TR\_PROTFLT\_ADDR event as well as their associated data in the message display area can be seen in Figure 13-9.



**Figure 13-9. TR\_PAGEFLT\_ADDR and TR\_PROTFLT\_ADDR Events**

Note the TR\_PROTFLT\_ADDR event to the left of the current time line at time=9.459738 and the TR\_PAGEFLT\_ADDR event to the right of the current time line at time=9.460050 and the corresponding data in the message display area. (See Chapter 9 for more information on the message display area and other elements of the display page.)

Note also that the TR\_PROTFLT\_ADDR and TR\_PAGEFLT\_ADDR events are represented by a vertical line that only intersects the exception state graph whereas a TR\_SWITCHIN event (see “Context Switch Trace Event” on page 13-2) intersects both the exception and syscall state graphs. In addition, TR\_PROTFLT\_ADDR and TR\_PAGEFLT\_ADDR events will only appear within a currently executing exception. This can be seen in Figure 13-10.

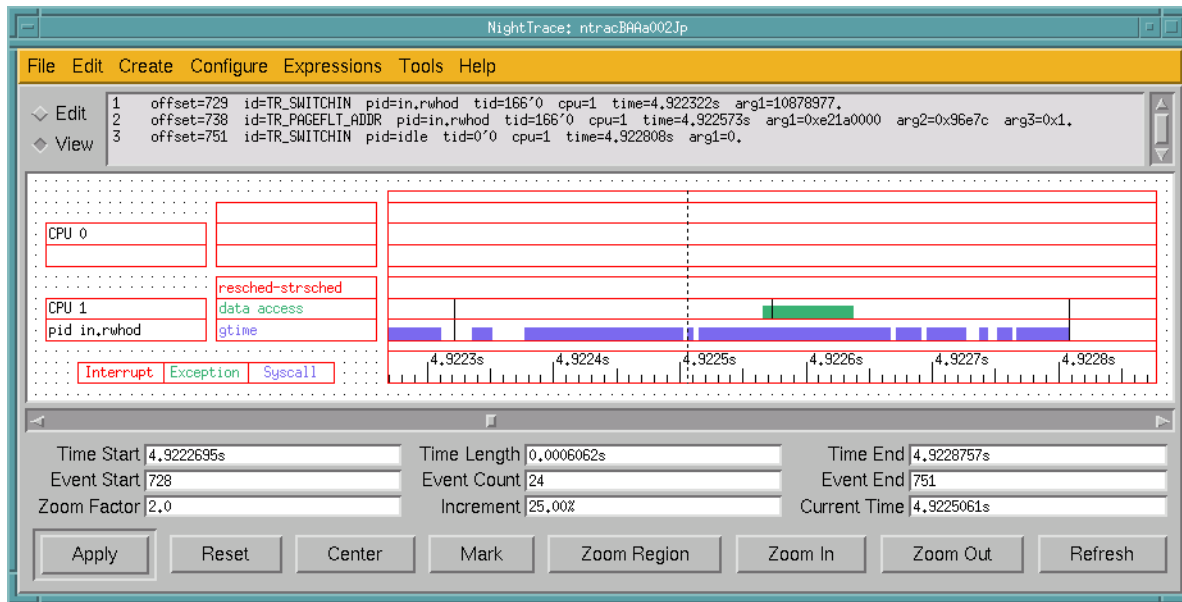


Figure 13-10. TR\_SWITCHIN vs. TR\_PAGEFLT\_ADDR and TR\_PROTFLT\_ADDR Events

## Syscall Information

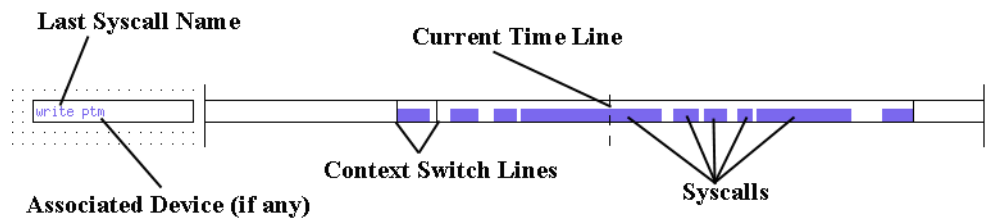


Figure 13-11. Last Syscall Box and Syscall Graph

Figure 13-11 shows a last syscall box and a syscall graph. The syscall graph displays a state that is drawn whenever a system call (syscall) is executing on the associated CPU. Unlike interrupts, syscalls cannot nest, so they are always graphed with the same height.

Context switch lines are superimposed on syscall graphs. It is common to see a context switch line at what looks like the very end (or beginning) of a syscall. Usually, this does not indicate that the syscall has ended, only that it has been suspended because the process that originated the syscall has switched out. The syscall resumes when the process is switched back in again. An example of a syscall being suspended and resumed can be seen at the right end of the syscall graph in Figure 13-11.

The last syscall box displays the last syscall prior to the current time line that executed (and may still be executing) on the associated CPU. If the syscall is associated with a device, the name of the device is shown after the name of the syscall.

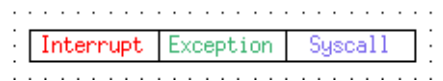
The last syscall box can be used with the syscall graph to identify any syscalls that are currently visible on the graph. Simply move the current time line onto a graphed syscall, and the last syscall box will update to display the name of the syscall.

Because the last syscall box displays the name of the last syscall that executed, it is possible for there to be no syscalls visible on the syscall graph even though the last syscall box contains a valid syscall name. This just signifies that the last syscall on the CPU ended prior to the beginning of the current interval.

It is possible for the first syscall logged by a process since kernel tracing began to be unknown. This can occur if the process is switched in and immediately resumes a syscall that was previously suspended. If this occurs, the last syscall box will display “can’t determine” for the name of the syscall.

The last syscall box is a Data Box (see “Data Box” on page 10-5), and the last syscall graph is a State Graph (see “State Graph” on page 10-7). See “Configuring Display Objects” on page 10-15 for more information on configuring Data Boxes and State Graphs.

## Color Information



**Figure 13-12. Color Key**

Figure 13-12 shows the color key that is located on the bottom left of the grid on the pre-defined kernel display pages. The color key is useful only on X terminals that support more colors than just black and white.

The text in the color key is color-coded. By default, the word “Interrupt” is red, and all display objects on the kernel display page that display information about interrupts are also red. By default, the word “Exception” is green, and all display objects that display information about exceptions are also green. By default, the word “Syscall” is blue, and all display objects that display information about syscalls are also blue.

The default colors of the different groups of kernel objects can be controlled with X resources. The colors are specified on a per-CPU basis. The default resources for logical CPU 0 are:

```
Ntrace*Color*GridObject*interrupt0*foreground: red
Ntrace*Color*GridObject*exception0*foreground: green
Ntrace*Color*GridObject*syscall0*foreground: blue
```

See Appendix B for more information on X resources.

## Kernel String Tables

There are seven kernel related pre-defined string tables. They are:

`vector` This string table contains the interrupt and exception vector names associated with the system that the kernel tracing was performed on. It is contained in the `vectors` file.

This table is indexed by an exception/interrupt vector number or an exception/interrupt vector name. Examples of using this table are:

```
get_string(vector, arg3())
get_string(vector, 15)
get_item(vector, "ncr_intr")
```

`syscall` This string table contains the names of all the possible syscalls that can occur on the system. It is contained in the `vectors` file. For brief descriptions of the entries in the `syscall` table, see "Syscalls" on page 13-18.

This table is indexed by a system call number or a system call name. Examples of using this table are:

```
get_string(syscall, 44)
get_string(syscall, arg2())
get_item(syscall, "fork")
```

`device` This string table contains the names the devices that are currently configured in the kernel. It is contained in the `vectors` file.

This table is indexed by a device number or a device name. Examples of using this table are:

```
get_string(device, arg3())
get_string(device, 720900)
get_item(device, "gd")
```

`name_pid` This string table contains the name of each node's process ID table. It is dynamically built as the trace event files are processed upon initialization.

`node_name` This string table contains the names of all nodes that have a trace event file associated with them. It is dynamically built as the trace event files are processed upon initialization.

`pid_nodename` This string table contains the names associated with all process identifiers found in trace event files for node name *nodename*. It is dynamically built as the trace event files are processed upon initialization. It is contained in the `vectors` file. Because process identifiers are not guaranteed to be unique across nodes, using the predefined string table `pid` to get the process name for a process ID may result in an incorrect name being returned from the table. Using the node process ID tables ensures that the correct process name is returned for a process ID unless the process name is not unique on that particular node.

These tables are indexed by a process identifier or a process name. Examples of using these tables are:

```
get_string(pid_hal, pid())
get_item(pid_simulator, "odyssey")
```

`syscall_nodename` This string table contains the names of all possible system calls that can occur in trace event files for node name *nodename*. It is contained in the vectors file.

This table is indexed by a system call number or a system call name. Examples of using this table are:

```
get_string(syscall_systemx, 31)
get_string(syscall_systemy, arg2())
get_item(syscall_systemz, "read")
```

`vector_nodename` This string table contains the interrupt and exception vector names associated with trace event files for node name *nodename*. It is contained in the vectors file.

This table is indexed by an exception/interrupt vector number or an exception/interrupt vector name. Examples of using this table are:

```
get_string(vector_machine1, arg3())
get_string(vector_machine2, 585)
get_item(vector_system3, "data access")
```

`device_nodename` This string table contains the names of devices configured in the kernel for trace event files from node name *nodename*. It is contained in the vectors file.

This table is indexed by a device number or a device name. Examples of using this table are:

```
get_string(device_simulator1, arg3())
get_string(device_simulator4, 3604484)
get_item(device_controller, "rtc")
```

The `pid` string table is also used by the kernel display pages. For more information on the `pid` string table, see “Pre-Defined String s” on page 4-16.

## Kernel Reference

The following sections provide a brief reference to the most common interrupts, exceptions, and syscalls.

### Interrupts

There are many different types of interrupts that can be logged by the kernel. The possible types are listed in the system-dependent `vector` string table in the vectors file. There are two main categories of interrupts:

- Non-device-related interrupts
- Device-related interrupts

The members of these two categories are described in the following two sections.

## Non-Device-Related Interrupts

Table 13-2 provides an alphabetical list of the most common non-device-related interrupts.

**Table 13-2. Non-Device-Related Interrupt Reference**

Interrupt	Description
callout int	A real time clock interrupt that is used internally by the kernel.
console wake	An interrupt caused by the console wakeup button.
int on no int	An interrupt that occurs during the processing of another interrupt.
power fail	A power fail interrupt.
rescheduling	A rescheduling interrupt used to trigger a context switch to run the highest priority process that is ready to run.
softclock	An interrupt used to process system callout queue entries.
spurious int	An interrupt that usually indicates an unreported or already-removed interrupt. This interrupt appears only in kernel traces.
sysfault int	An interrupt indicating that a fatal hardware condition has been detected.
user int	A user-level interrupt. See <b>iconnect (3C)</b> for a description of enabling user-level interrupts.
xcall int	An inter-processor interrupt used for cache flushing, delivering exceptions to another processor, performance monitoring, and halting processors.

For more information about interrupts see **intstat (1M)** and **uistat (1M)**.

## Device-Related Interrupts

The names printed for device interrupts correspond to the device names in the system configuration files. See *System Administration Volume 2* for information on adding devices to a system.



Table 13-3 provides an alphabetical list of the most common device-specific interrupts. For more information on a device-specific interrupt, refer to the documentation associated with the particular device.

**Table 13-3. Device-Related Interrupt Reference**

Interrupt	Description
consintr	A console terminal interrupt.
eg	An Eagle ethernet controller interrupt.
eti_intr	An edge-triggered interrupt.
ex	An Excellan ethernet controller interrupt.
gpib	An IEEE-488 GPIB controller interrupt.
hardclock	A 60-Hertz clock interrupt.
hd	An HDC disk-controller interrupt.
hps	An HPS serial line-controller interrupt.
hrm	A reflective memory interrupt.
hsa	An HSA disk controller interrupt.
hsd	An HSD controller interrupt.
ie	An integral ethernet interrupt.
is	An integral SCSI controller interrupt.
mpcc	An MPCC controller interrupt.
pgintr	An FDDI controller interrupt.
rtcintr	A real-time clock interrupt.
xy	A Xylogics tape-controller interrupt.

## Exceptions

There are many different types of exceptions that can be logged by the kernel. The possible types are listed in the system-dependent `vector` string table in the `vectors` file.

Table 13-4 is an alphabetical list of the most common exceptions. See the *PowerPC 604 RISC Microprocessor User's Manual* for more information.

**Table 13-4. Exception Reference**

Exception	Description
data access	An exception indicating that a page fault for a data page occurred.
decrementer	An exception that occurs when the decrementer register counts down to zero.
float unavail	An exception that occurs the first time a process attempts to use the floating-point unit.
inst access	A page fault exception that occurs during an instruction fetch.
inst brkpt	An exception indicating that a breakpoint instruction was executed.
kstack overflow	A fatal exception generated due to kernel errors.
machine check	A fatal exception generated for various reasons including parity errors, hardware failures, and kernel errors.
misaligned	An exception indicating that a load, store, or exchange instruction was attempted with a destination memory address not consistent with the size of the access.
program	An exception indicating one of several possible conditions including divide by zero, invalid instruction, and privilege violation.
trace	An exception generated during single stepping of the CPU.

## Syscalls

The list system calls can be found in the architecture-dependent `syscall` string table that is dynamically generated into the vectors file.

# Performance Tuning

Although NightTrace's defaults are designed for maximum efficiency, your NightTrace environment and application may have special requirements that warrant some performance tuning. You may want to investigate the following issues:

- Preventing trace event loss
- Ensuring accurate timings
- Optimizing file system and CPU usage
- Conserving disk space
- Conserving memory and accelerating **ntrace**

## Preventing Trace Events Loss

By default, NightTrace copies all user trace events from the shared memory buffer to the trace event file. This means that normally NightTrace neither discards nor loses trace events.

To conserve disk space, you may invoke **ntraceud** with the **-filewrap** or **-buffer-wrap** option. However, by doing so, you are telling NightTrace to intentionally discard older or less-vital trace events. If discarding trace events is undesirable, run **ntraceud** in expansive mode. To do this, invoke **ntraceud** without the **-filewrap** and **-buffer-wrap** options. See “Conserving Disk Space” on page A-4 for more information.

When NightTrace *discards* trace events, it is intentional. When NightTrace *loses* trace events, it is not. NightTrace does not report discarded trace events; it does, however, report lost trace events. Most trace event loss is preventable by flushing the shared memory buffer often.

NightTrace shows trace event loss in the following ways:

- As a non-zero “events lost” statistic from **ntraceud -stats trace\_file**, from **ntrace -filestats**, or on the **ntrace** Global Window
- As a reverse video “L” on the **ntrace** display page Ruler at the location where the trace event was lost

If trace event loss seems excessive, you can do the following:

Action	Reason
Decrease <b>-cutoff</b> , the shared memory buffer-full cutoff percentage for <b>ntraceud</b>	Increase the chance that the <b>ntraceud</b> daemon will have enough time to copy the trace events in the shared memory buffer to disk before the shared memory buffer fills up.
Decrease <b>-timeout</b> , the <b>ntraceud</b> timeout interval	(Same)
Call <code>trace_flush()</code> or <code>trace_trigger()</code> often from within your application, especially when your application is at a non-time critical point	(Same)
Increase <b>-memsize</b> , the shared memory buffer size for <b>ntraceud</b>	(Same)

Use the following command to see the system settings for the current, default, minimum, and maximum shared memory segment size:

```
$ /etc/conf/bin/idtune -g SHMMAX
```

See the **idtune (1M)** man page for more information.

A few other factors can affect trace event loss. Processes in your application may write trace events into the shared memory buffer at the same time that **ntraceud** is flushing trace events from the shared memory buffer to the trace event file; if the trace event incoming rate exceeds the flush rate, trace events may not be recorded. Furthermore, when NightTrace must choose between operating unobtrusively and logging all trace events, it favors being unobtrusive.

See Chapter 6 for more information on **ntraceud** options and modes. For more information on `trace_flush()` or `trace_trigger()`, see “`trace_flush()` and `trace_trigger()`” on page 2-21.

If events are being lost during kernel tracing:

- Verify that the output KernelTrace trace event file is on a local file system and not an NFS file system. If you run the following command and there is a colon (:) in the “Filesystem” column, the file is on an NFS file system.

```
$ df kernel_trace_file
```

- Ask your system administrator to increase the size of `TR_BUFFER_COUNT` in `/etc/conf/mtune.d/trace` by running the **idtune (1M)** command, rebuild, and reboot the system. (Usually a `TR_BUFFER_COUNT` of 5 is sufficient.) The kernel allocates buffers of 3 pages each (12,288 bytes) for kernel tracing. This is part of the kernel’s initialized global data, meaning these are reserved physical pages.

## Ensuring Accurate Timings

If you lack the privilege to lock your pages in memory (`P_PLOCK`), you must invoke `ntraceud` with the `-lockdisable` option. If your application lacks read and write privilege to `/dev/spl` you must invoke `ntraceud` with the `-ipldisable` option. Invoking `ntraceud` with either the `-lockdisable` or `-ipldisable` option, may introduce delays and waiting within your application. Use the `-lockdisable` and `-ipldisable` options only when necessary. For more information on the `-lockdisable` option, see “Option to Prevent Page Locking (`-lockdisable`)” on page 6-11. For more information on the `-ipldisable` option, see “Option to Disable the IPL Register (`-ipldisable`)” on page 6-9.

By default, `ntraceud` and NightTrace library routines use page locking to prevent page faults during trace event logging. NightTrace also modifies the interrupt priority level (IPL) register; this action prevents rescheduling and interrupts during trace event logging. NightTrace prevents the operating system from pre-empting your trace event logging application to make itself most unobtrusive to your application.

If the application must wake the `ntraceud` daemon unexpectedly, overhead can cause trace event timings to be distorted. Do one or more of the following to increase the likelihood that the daemon will be awake when needed and to make sure that disk write rates are as fast as the application’s logging rate:

- Increase the shared memory buffer size (`-memsize`)
- Decrease the shared memory buffer-full cutoff percentage (`-cutoff`)
- Decrease the `ntraceud` timeout interval (`-timeout`)
- Call `trace_flush()` or `trace_trigger()` appropriately

For more information on the `-memsize`, `-cutoff`, and `-timeout` options, and `trace_flush()`, see, respectively, “Option to Define Shared Memory Buffer Size (`-memsize`)” on page 6-16, “Option to Set the Buffer-Full Cutoff Percentage (`-cutoff`)” on page 6-18, “Option to Set Timeout Interval (`-timeout`)” on page 6-17, and “`trace_flush()` and `trace_trigger()`” on page 2-21.

## Optimizing File System and CPU Usage

Different systems may share files via the Network File System (NFS); however, accessing an NFS-mounted file takes longer than accessing a local file. You get the best NightTrace and KernelTrace performance if you avoid NFS accesses; put your trace event file on the same system where both the daemons and your application run. To determine whether your disk is local to your system, verify that it is mounted on `/dev` and not on another host. You can do this by running the `df (1)` command and looking for a colon (:) in the “Filesystem” column.

A single system may have more than one CPU. Consider assigning the daemon and your application to different CPUs on the same system; this way, the daemons will not interfere with your application.

You can use the **mpadvise (3C)** library routine to help you determine which CPUs exist on this system. You can trace the daemon and your application to particular CPUs with the **run (1)** command.

```
$ run -bbias command
```

## Conserving Disk Space

To determine how much disk space is available on your system, run the **df (1)** command with the **-k** option and look at the “avail” column. You can conserve disk space if you permit NightTrace to discard some trace events. To do this, invoke **ntraceud** with either the **-filewrap** option or the **-bufferwrap** option.

The **ntraceud -filewrap** option makes NightTrace operate in file-wraparound mode, rather than in expansive mode. In file-wraparound mode the trace event file can become full of trace events. When this happens, **ntraceud** overwrites the oldest trace events at the beginning of the file with the newest ones. The overwriting is called *discarding trace events*. For more information on file-wraparound mode, see “Option to Establish File-Wraparound Mode (-filewrap)” on page 6-12.

The **ntraceud -bufferwrap** option makes NightTrace operate in buffer-wraparound mode, rather than in expansive mode. When the buffer is full in buffer-wraparound mode, the application treats the shared memory buffer as a circular queue and overwrites the oldest trace events with the newest ones. This overwriting continues until your application explicitly calls `trace_flush()` or `trace_trigger()`. Only then, does **ntraceud** copy the remaining trace events from the shared memory buffer to the trace event file. The overwriting is called *discarding trace events*. For more information on buffer-wraparound mode, see “Option to Establish Buffer-Wraparound Mode (-bufferwrap)” on page 6-13.

By default, **ntraceud** operates in expansive mode, not file-wraparound or buffer-wraparound mode. In expansive mode, NightTrace uses the most disk space because it does not discard any trace events.

You can also conserve disk space by invoking **ntraceud** with the **-disable** option so it logs fewer trace events. For details, see “`trace_enable()`, `trace_disable()`, and Their Variants” on page 2-17.

## Conserving Memory and Accelerating ntrace

**ntrace** can be a memory-intensive tool. By default, when **ntrace** starts up, it loads all trace event information into memory; therefore, the more trace events in your trace event file(s), the more memory **ntrace** uses. When you move the scroll bar on the Display Page to change the displayed interval, **ntrace** processes all trace events between the last interval and this one; if there are many trace events, the display update (or search) may seem slow. To conserve memory and accelerate **ntrace**:

- Log only trace events you are really interested in.

- Invoke **ntrace** only with the trace event files that are essential to your analysis.
- Invoke **ntrace** with options (**-nohardclock**, **-process -start**, and **-end**) that restrict which trace events get loaded. For more information about **ntrace** options, see “Command-line Options” on page 4-1.





## GUI Customization

The graphical user interface (GUI) for **ntrace** is based on OSF/Motif. **ntrace** runs in the environment of the X Window System. Your X terminal vendor supplies you with vendor-specific directories and files that pertain to colors and fonts. The file that contains available colors is called **rgb.txt**. The fonts that your X server supports are in the **/usr/lib/X11/fonts** directory.

NightTrace resources are found in **/usr/lib/X11/app-defaults/Ntrace**. These resources are used by all dialogs, windows, display pages, etc.

**ntrace** has default values for X resources. These resources include fonts, some push button names, window titles, window-component dimensions, and colors. You can override the following default X resource settings by providing new values in the following places:

- In your **.Xdefaults** file
- On the **ntrace** invocation line
- In a resource file that the **xrdb(1)** X resource database manager reads

If you specify the same X resource on the **ntrace** invocation line and in your **.Xdefaults** file, the setting on the invocation line overrides the one in the file.

An X resource line has the following format:

```
object*subobject [*subobject . . . ]*attribute: value
```

where:

<i>object</i>	Is the name of the X client program, <b>Ntrace</b> .
<i>subobject</i>	Is a level in the widget (window component) hierarchy with the most general level first; this always begins on an upper-case letter.
<i>attribute</i>	Is a characteristic of the last <i>subobject</i> ; this always begins on a lower-case letter.
<i>value</i>	Is a setting for the <i>attribute</i> .

It is possible to omit levels from the widget hierarchy. If you specify all levels of the widget hierarchy and then a *value*, the value applies to that specific widget. If you leave out levels of the widget hierarchy, the attribute applies more generally, possibly to a class of widgets.

For more information on X resources, see “Recommended Reading” on page 1-6 and the *X Window System User’s Guide*.

## Default X-Resource Settings for ntrace

**ntrace**'s default X-resource settings follow. They are primarily grouped by window and display object. There are some subobjects and attributes that appear in many settings.

In the following X-resource strings, default values are shown where they exist.

The resource strings for grid attributes follow. **ntrace** uses the `defaultDotsHigh` and `defaultDotsWide` attributes only for new display pages. Note: if `defaultDotsWide` is too narrow to accommodate all the display page push buttons, **ntrace** overrides this setting.

```
Ntrace*Grid*foreground:  
Ntrace*Grid*background:  
Ntrace*Grid*font:  
Ntrace*Grid*defaultDotsHigh: 30  
Ntrace*Grid*defaultDotsWide: 60
```

### TIP:

Experiment with colors and shadings until you find a set you like. To avoid visual fatigue, use highly-contrasting colors and values sparingly.

The resource strings for the specific display objects are:

```
Ntrace*Color*GridLabel*background:  
Ntrace*Color*GridLabel*foreground:  
Ntrace*Color*GridLabel*font:  
Ntrace*Color*GridLabel*textJustify:  
Ntrace*Color*GridLabel*textGravity:
```

```
Ntrace*Color*DataBox*background:  
Ntrace*Color*DataBox*foreground:  
Ntrace*Color*DataBox*font:  
Ntrace*Color*DataBox*textJustify:  
Ntrace*Color*DataBox*textGravity:
```

```
Ntrace*Color*Column*background:  
Ntrace*Color*Column*foreground:
```

```
Ntrace*Color*StateGraph*background:  
Ntrace*Color*StateGraph*foreground:  
Ntrace*Color*StateGraph*eventColor:
```

```
Ntrace*Color*EventGraph*background:  
Ntrace*Color*EventGraph*foreground:
```

```
Ntrace*Color*DataGraph*background:  
Ntrace*Color*DataGraph*foreground:
```

```
Ntrace*Color*Ruler*background:  
Ntrace*Color*Ruler*foreground:  
Ntrace*Color*Ruler*font:  
Ntrace*Color*Ruler*markColor:  
Ntrace*Color*Ruler*lostEventColor:
```

Grid object settings apply if you have not set the corresponding setting for a specific display object. The general grid object resource strings are:

```
Ntrace*Color*GridObject*background:
Ntrace*Color*GridObject*foreground:
Ntrace*Color*GridObject*borderColor:
```

For information about setting X resources for kernel displays, see “Color Information” on page 13-13.

## Examples

Setting X resources to values is most consistent if the values of the X resources do not contain spaces. For example, even if your `rgb.txt` color file contains a color called “navy blue,” for simplicity type it as one word without any quotation marks.

In the following examples, you are making navy blue (`navyblue`) the foreground color (`foreground`) of all grid objects (`GridObject`) on a color monitor (`Color`) for `ntrace` (`Ntrace`). This example shows how this line may appear in your `.Xdefaults` file.

```
Ntrace*color*GridObject*foreground: navyblue
```

The following example shows how you can use this setting on the `ntrace` invocation line. Note: there must not be any spaces between the colon and the value.

```
$ ntrace -xrm Ntrace*color*GridObject*foreground:navyblue
```

## Exercise: Customizing Display Colors

Edit your `.Xdefaults` file so it defines background colors for the following display objects. Suggested colors are provided.

**Table B-1. Suggested Colors for X Resources**

Display Object	Suggested Color
Column	CornflowerBlue
Data Graph	PowderBlue
State Graph	LightSteelBlue
Ruler	PaleGreen
Data Box	Aquamarine
Grid Object	SkyBlue

A possible solution follows:

```
Ntrace*Color*Column*background: CornflowerBlue  
Ntrace*Color*DataGraph*background: PowderBlue  
Ntrace*Color*StateGraph*background: LightSteelBlue  
Ntrace*Color*Ruler*background: PaleGreen  
Ntrace*Color*DataBox*background: Aquamarine  
Ntrace*Color*GridObject*background: SkyBlue
```

To test your entries at an X terminal, invoke **ntrace** with the **log** trace event file, and bring up the default display page.

## Answers to Common Questions

---

**Q:** What can I do if trace events are not logging at all?

**A:** Verify that the trace event file name on the `trace_begin()` call matches the one on the user daemon invocation. Furthermore, check that the file exists and that you have permission to read and write it. Additionally, be sure your thread name contains no embedded spaces or punctuation, including periods. See “`trace_begin()`” on page 2-5 and “`trace_open_thread()`” on page 2-10 for more information.

**Q:** When should I log a different trace event ID number?

**A:** Each endpoint of a state should have a different trace event ID number. Usually each trace event logging routine logs a different trace event ID number. This lets you easily identify which source line logged the trace event, how often that source line executed, and what order source lines executed in. However, it is sometimes useful to log the same trace event ID in multiple places. This makes it possible to group trace events from related, but not identical, activities. For more information, see “`trace_event()` and Its Variants” on page 2-11.

**Q:** How can I prevent user trace events from being discarded or lost?

**A:** Use expansive mode; avoid use of buffer or file wrapping options. Flush the shared memory buffer more often by tuning:

- The shared memory buffer size
- The shared memory buffer’s flush percentage
- Increase the priority of the user trace daemon
- Bind the user trace daemon to a CPU with minimal activity

See “Preventing Trace Events Loss” on page A-1 and Chapter 6 for more information.

**Q:** What can I do if trace events are not appearing in an ntrace display?

**A:** Press **Refresh**, fill out the Search Form, fill in values in the interval control area, use the interval scroll bar, keep pressing the **Zoom Out** push button until you see trace events, examine a display object configuration so you know what it is “listening” for, add or reconfigure display objects on the grid. See Chapter 8 “Viewing Trace Event Logs” for more information.

**Q:** My trace event timings occasionally have huge gaps of time between them. What is the cause?

**A:** You are probably running your application on a Series 6000 system and are calling `clock_settime()`. This system call can corrupt the system interval timer which NightTrace uses for trace event timings.

**Q:** How can I prevent kernel trace events from being lost?

**A:**

- Verify that the raw kernel trace output file is on a local file system and not an NFS file system.
- Ask your system administrator to increase the size of `TR_BUFFER_COUNT` kernel tunable parameter (PowerMAX Only)
- Increase the priority of the kernel trace daemon
- Bind the kernel trace daemon to a CPU with minimal activity

# Glossary

---

This glossary defines terms used in the documentation. Terms in *italics* are defined here.

## Ada task

An Ada task is a construct of statements which logically execute in parallel with other tasks within an Ada program (process). Tasks communicate asynchronously via variables whose visibility is defined by normal Ada scoping rules. Tasks communicate synchronously via rendezvous between a calling and accepting task.

## Add

A *push button* that creates a new *macro*, *qualified event*, or *qualified state* on the current *display page*.

## Apply

A *push button* that validates and saves all changes. The same functionality is available by pressing <Enter> in a modified field.

## argument

See *trace event argument*.

## boolean table

A pre-defined *string table* which associates 0 with `false` and all other values with `true`.

## buffer-wraparound mode

The mode that causes the `ntraceud` daemon to treat the *shared memory buffer* as a circular queue and to overwrite the oldest *trace events* with the newest ones; this means that `ntraceud` intentionally discards the oldest trace events to make room for the newest ones. Invoke `ntraceud` with the `-bufferwrap` option to obtain this behavior. The two other `ntraceud` modes are *expansive mode* and *file-wrap-around mode*.

## button

See *mouse button*, *push button*, and *radio button*.

## click

To press and release a *mouse button* without moving the pointer. Usually you do this in NightTrace to select menu items, *push buttons*, or *radio buttons*.

**Close**

A *push button* that closes a *dialog box*. This can also be a menu item that makes a *window* close.

**Column**

A *display object* that constrains the width of *State Graphs*, *Event Graphs*, *Data Graphs*, and *Rulers*.

**configuration**

The definition of a *display object*, *macro*, *qualified event*, or *qualified state*.

**configuration file**

An NightTrace-generated ASCII file that holds *display pages*, *macro*, *qualified event*, and *qualified state* definitions. This can also be a hand-edited table file, containing definition of *string tables* and/or *format tables*.

**Configure**

A *push button* that reconfigures and renames the selected *macro*, *qualified event*, or *qualified state*.

**context switch**

An action that occurs inside the kernel. Its functions are to save the state of the process that is currently executing, to initialize the state of the process to be run, and to begin execution of the new process.

**context switch line**

A vertical line superimposed on an *exception graph* or a *syscall graph* on a kernel *display page*. It indicates that the kernel has switched out the process that was previously running on the CPU and switched in a new process.

**control**

See *mouse button*, *push button* and *radio button*.

**CPU box**

A *Grid Label* on a kernel *display page*. It identifies which logical central processing unit the displayed data corresponds to. Logical CPU numbers are related to, but not necessarily identical to, physical CPU numbers.

**current instance of a state**

The instance of a *state* which has begun but has not yet completed. Thus, the *current time line* would be positioned within the region from the start event up to, but not including, the end event.



**current time**

The time in the *interval* up to which all *display objects* on a *display page* have been updated.

**current time line**

The dashed vertical bar that represents the *current time* in a *Column*.

**current trace event**

The last *trace event* on or before the *current time line*.

**cursor**

See *text cursor*.

**daemon definition**

The configuration of a particular trace daemon which includes daemon collection modes and settings, daemon priorities and CPU bindings, and data output formats, as well as which trace event types are handled by that daemon.

**Data Box**

A *display object* that displays possibly variable textual or numeric information.

**Data Graph**

A scrollable *display object* that graphically displays a bar chart of an *expression*'s value as it changes over the *interval*.

**Default Kernel Page**

A menu item that automatically creates a *display page* to depict *context switches*, *interrupts*, *exceptions*, and system calls with *display objects* for each CPU on the system.

**Default Page**

A menu item that automatically creates a *display page* with a *State Graph* for each trace event logging process in your *trace event file(s)*.

**Delete**

Remove the selected *macro*, *qualified event*, or *qualified state*.

**device table**

A pre-defined, dynamically generated *string table* in the **vectors** file created by **ntrace** when consuming raw kernel trace data files. string table contains the names of the devices that are currently configured in the kernel.

**dialog box**

A transient secondary *window* that accepts input or conveys a message, for example information, errors, warnings, and questions. This construct is occasionally called a pop-up window.

**dimmed**

See *disabled*.

**disabled**

To flag a component, such as a menu item or *push button*, as temporarily unavailable by graying out the label.

**discarded trace event**

A *trace event* that **ntraceud** intentionally did not log in *buffer-wraparound* or *file-wraparound mode*.

**display object**

A user-configured graphical component of a *display page* that shows *trace events*, *states*, *trace event arguments*, other numeric and text data. Display objects include the following: *Grid Labels*, *Data Boxes*, *Columns*, *State Graphs*, *Event Graphs*, *Data Graphs* and *Rulers*.

**display page**

The NightTrace *window* that allows you to layout *display objects* and see *trace event* and *state* information in them. You can store display pages in *configuration files*.

**dotted area**

See *grid*.

**drag**

To press and hold down a *mouse button* while moving the *mouse*. Usually you do this in NightTrace to position a *display object*.

**duration**

The period of time between the start and end *trace events* of some *state*.

**Edit mode**

The *display-page* mode that allows you to create, edit, and configure *display objects*, *macros*, *qualified events*, and *qualified states*. The other display-page mode is *View mode*.

**ellipses (...)**

An indicator at the end of a menu item that tells you this selection makes a *dialog box* appear. Also, an indicator in command line option summaries and syntax listings that tells you more than one occurrence of the previous syntactic component is allowed.

**end function**

A *state function* that provides information about the ending *trace event* of the *last completed instance of a state*. The *state* to which the end function applies is either the *qualified state* specified to the *function*, or the state being currently defined. Thus, if a qualified state is not specified, end functions are only meaningful when used in *expressions* associated within a state definition.

**event**

See *trace event*.

**event\_arg\_dbl\_summary table**

A pre-defined *format table* which contains formats for statistical displays of trace event *matches* and type double *arguments*.

**event\_arg\_summary table**

A pre-defined *format table* which contains formats for statistical displays of trace event *matches* and type long *arguments*.

**Event Graph**

A scrollable *display object* that graphically displays *trace events* as vertical lines in a *Column*.

**event ID**

See *trace event ID*.

**event map file**

User-generated ASCII file that lets you associate or map short mnemonic names with numeric *trace event IDs*.

**event\_summary table**

A pre-defined *format table* which contains formats for statistical displays of trace event *matches* and trace event time *gaps*. It determines the default event-summary output format.

**event table**

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and maps all known numeric *trace event IDs* with symbolic trace event names.

## exception

An event internal to the currently executing process that stops the current execution stream. Exceptions can be suspended and resumed.

## exception graph

A *State Graph* on a kernel *display page*. It displays *states* representing *exceptions* executing on the associated CPU.

## expansive mode

The (default) mode that causes the **ntraceud** daemon to copy all *trace events* that ever reach the *shared memory buffer* to the indefinitely-sized *trace event file*. Invoke **ntraceud** without the **-filewrap** and **-bufferwrap** options to obtain this behavior. The two other **ntraceud** modes are *buffer-wraparound mode* and *file-wraparound mode*.

## expression

A combination of operators and operands that evaluate to a value. Operands include constants, *macro* calls, *function* calls, *qualified events*, and *qualified states*.

## Exit

A menu item that terminates an NightTrace session.

## file-wraparound mode

The mode that causes the **ntraceud** daemon to overwrite the oldest *trace events* in the beginning of the *trace event file* with the newest ones; this means that **ntraceud** intentionally *discards* the oldest trace events to make room for the newest ones. Invoke **ntraceud** with the **-filewrap** option to obtain this behavior. The two other **ntraceud** modes are *expansive mode* and *buffer-wraparound mode*.

## flushing the buffer

The process of the **ntraceud** daemon copying *trace events* from the *shared memory buffer* to a *trace event file*.

## font

A style of text characters.

## format function

A *function* that allows you to display a string.

## format table

The pre-defined or user-defined structure that allows you to group related integer values together and associate each one with a corresponding dynamically-formatted

and generated character string. You hand-edit format tables into *configuration files*. The related structure is a *string table*.

**function**

A pre-defined NightTrace entity that may be used in an *expression*. NightTrace provides several classes of functions: *trace event*, *multi-event*, *start*, *end*, *multi-state*, *offset*, *summary*, *format*, and *table functions*.

**gap**

The period of time between two *trace events*, possibly the end of one *state* and the beginning of another.

**global process identifier**

See *PID*.

**Global Window**

The NightTrace *window* that displays summary statistics pertaining to your *trace event files* and allows you to open NightTrace-related files.

**graphical user interface**

The mechanism NightTrace uses to receive input and provide displays. It is based on the X Window System and Motif.

**grid**

The region of the *display page* filled with parallel rows and columns of dots that holds *display objects*.

**Grid Label**

A *display object* that displays constant textual information.

**GUI**

See *graphical user interface*.

**Help**

A menu item that presents the online manual using the HyperHelp viewer.

**host system**

The system on which the NightTrace GUI is running.

## icon

The small graphical image and/or text label that represents a *window* or window family when the window is minimized. The text label is either the window title or an abbreviated form of the title. Iconified windows are still active.

## ID

See *trace event ID*.

## instrumented code

Source code after you have put calls to NightTrace library routines into it.

## interrupt

An event external to the currently executing process; an interrupt stops the current execution stream to begin execution of a higher-priority execution stream. There are device-related and software-generated interrupts. Interrupts have an associated priority known as the interrupt priority level (IPL), which allows an interrupt to interrupt the execution stream of a lower-IPL interrupt.

## interrupt graph

A *Data Graph* on a kernel *display page*. It displays *states* representing *interrupts* executing on the associated CPU.

## interrupt priority level (IPL) register

A system register than can be used by the NightTrace library to prevent rescheduling and interrupts during trace event logging.

## interval

A time period in the trace session delimited by the **Start Time** and **End Time** fields of the *interval control area*.

## interval control area

The region of the *display page* that holds nine numeric fields that define and manipulate the *interval* and the *display objects* on the *grid*.

## interval timer

The system timer on the NightHawk 6000 Series and TurboHawk systems that *NightTrace* uses to timestamp *trace events*.

## Kernel Trace Event File

A *trace event file* is generated by a kernel trace daemon. This file contains raw kernel data and is automatically transformed into a filtered file (with a new filename using the ".**ntf**" suffix) by **ntrace**. Either a raw kernel trace event file or a filtered file may be specified to **ntrace**. The filtering process also creates a vectors

file which is formed by appending a “.vec” suffix to the original trace event file name.

### keyboard

A traditional input device for entering text into fields. In this manual, this is a standard 101-key North American keyboard.

### last completed instance of a state

The most recent instance of a *state* that has already completed. Thus, the *current time line* would be positioned either on, or after, the *end event* for a state.

### last exception box

A *Data Box* on a kernel *display page*. It displays the last *exception* prior to the *current time line* that executed (and may still be executing) on the associated CPU.

### last interrupt box

A *Data Box* on a kernel *display page*. It displays the name of the last *interrupt* prior to the *current time line* that executed (and may still be executing) on the associated CPU.

### last syscall box

A *Data Box* on a kernel *display page*. It displays the last *syscall* prior to the *current time line* that executed (and may still be executing) on the associated CPU.

### lightweight process identifier

See *LWPID*.

### lost trace event

A *trace event* **ntraceud** was unable to log. Several **ntraceud** options exist to prevent this trace event loss.

### LWPID

An integer that represents an operating system lightweight process identifier. It makes up the second half of a *PID*.

### macro

A user-defined named *expression* stored in a *configuration file*. When you call a macro, precede the macro name with a dollar sign.

### mark

The solid triangle on a *Ruler* that points to a particular time.

**match**

A *trace event* or *state* that meets user-defined qualifying configuration criteria.

**menu**

A list of user-selectable choices.

**menu bar**

The horizontal band near the top of a *window* that contains a list of labeled *pull-down menus*.

**message display area**

The scrolling region of the *Global Window* or the *display page* that holds textual statistics, as well as error and warning messages.

**most recent instance of a state**

If the *current time line* is positioned within a *current instance of a state*, then it is that instance of the *state*. Otherwise, it is the *last completed instance of a state*.

**mouse**

In this manual, a three-button pointing device for point-and-click interfaces.

**mouse button**

A part of the *mouse* that you can press to alter aspects of the application. Each mouse button has a different purpose. Button 1 is usually for selecting or dragging. Button 2 is usually for moving *display objects*. Button 3 is usually for resizing display objects. You can make multiple selections by simultaneously pressing <Shift> and clicking mouse button 1. You may *click*, *drag*, *press*, and *release* mouse buttons.

**multi-event function**

Multi-event functions return information about occurrences of events, or relationships between occurrences of events, before the *current time line*.

**multi-state function**

Multi-state functions return information about instances of states, or relationships between instances of states, before the *current time line*.

**name\_pid table**

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and associates node ID numbers with the the name of each node's process ID table.



**name\_tid table**

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and associates node ID numbers with the the name of each node's thread ID table.

**New Page**

A menu item that creates an empty *display page*.

**NightTrace**

The interactive debugging and performance analysis tool that is part of the NightStar tool kit. It consists of the **ntraceud** daemon, NightTrace library routines, and the **ntrace** display utility. This product allows you to log *trace events* and data from applications written in C, Fortran, or Ada; these applications may be composed of one or more processes, running on one or more CPUs. You can then examine these trace events and those from the kernel through the **ntrace** display utility.

**NightTrace thread**

A process, *thread* or *Ada task* (or a set of any combination of these) that is associated with a uniquely named *trace context*. The thread name is derived from the argument specified to the `trace_open_thread()` function.

**NightTrace thread identifier**

See *TID*.

**NightView**

A symbolic debugger that is part of the NightStar tool kit. It lets you debug C and Fortran applications; these applications may be composed of one or more processes, running on one or more CPUs. Among other things, NightView can automatically patch trace event logging routines into your executable application.

**node**

A system from which a *trace event file* can come from.

**node box**

If the RCIM synchronized tick clock is used to timestamp events, this is a *Grid Label* on a kernel *display page*. It identifies which *node* to which the displayed data corresponds.

**node ID**

A unique identifier internally assigned by NightTrace to every *node* that has an *trace event file* in a trace file analysis.

**node name**

The name of a system from which a *trace event file* can come.

**node\_name table**

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and associates *node ID* numbers with *node names*.

**node PID table**

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and associates process identifiers (*PIDs*) with process names for a particular *node*. The name of each node's table is `pid_nodename` where *nodename* is the node's name. If kernel tracing, this table is stored in the **vectors** file.

**node TID table**

A pre-defined, dynamically generated *string table*. It is internal to NightTrace. If user tracing, it associates NightTrace thread ID numbers with thread names for a particular *node*. If kernel tracing, this table is not used. The name of each node's table is `tid_nodename` where *nodename* is the node's name.

**NT\_ASSOC\_PID**

An overhead *trace event* that **ntraceud** logs at the beginning and end of each process.

**NT\_ASSOC\_TID**

An overhead *trace event* that **ntraceud** logs at the beginning and end of each *thread* and *Ada task*.

**NT\_CONTINUE**

An overhead *trace event* that **ntraceud** logs for multi-argument trace events.

**ntrace display utility**

The part of *NightTrace* that graphically displays *trace events*, trace event data, and *states* for debugging and performance analysis.

**ntraceud**

The *NightTrace* daemon process that allows you to log user-defined *trace events* and data from user applications written in C, Fortran, or Ada. These applications may be composed of one or more processes, running on one or more CPUs.

**object**

See *display object*.

**offset**

The number that identifies the position of a *trace event* in the chronologically-ordered sequence of trace events, regardless of the *trace event ID*. Counting

starts from zero. For example, if a trace event with trace event ID 71 is the third trace event in the trace session, then its offset is 2.

**offset function**

A *function* that takes an *expression* that evaluates to an *offset* as a parameter.

**OK**

A *push button* that acknowledges the warning in a *dialog box*.

**Open**

A menu item and *push button* that opens an existing file.

**ordinal trace event number**

See *offset*.

**panel**

A *window* component that groups related buttons, for example *push buttons*.

**PID**

A 32-bit integer that represents an operating system process. The following syntax numerically specifies a PID: *raw\_PID*·*LWPID*. The operating system process identifier (*raw PID*) is contained in the upper 16 bits and the lightweight process identifier (*LWPID*) is contained in the lower 16 bits.

**PID table**

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and associates process identifiers (*PIDs*) with process names. If kernel tracing, the `pid` string table in the **vectors** file.

**point**

To move the *mouse* so the mouse pointer is positioned at the place of interest.

**pointer**

A graphical symbol that represents the mouse pointer's current location in the *window*. The shape of the pointer shows the current usage. Usually a pointer is shaped like an arrow pointing to the upper left.

**pop-up window**

See *dialog box*.

**press**

To hold down a *mouse button* without releasing it or to depress a *keyboard key*.

**pull-down menu**

A list of related choices called menu items pulled down from the *menu bar*. *Click* on a menu item to select it.

**push button**

A graphic image of a labeled button. *Click* on a push button to select it.

**qualified event**

Qualified events provide a means for referencing a set of one or more trace events which may be restricted by conditions specified by the user.

**qualified state**

Qualified states provide a means for defining regions of time based on specific starting and ending events and restricted by conditions specified by the user.

**radio button**

A graphic, labeled diamond-shape that represents a mutually exclusive selection from related radio buttons. *Click* on a radio button to select it.

**raw PID**

A 16-bit integer that makes up the first half of a *PID*.

**RCIM**

The Real-Time Clock and Interrupt Module is a multi-function PCI mezzanine card (PMC) designed for time-critical applications that require rapid response to external events, synchronized clocks, and/or synchronized interrupts. The RCIM provides synchronized clocks (tick timer and posix format clock), edge-triggered interrupts, real-time clocks, and programmable interrupts.

**RCIM synchronized tick clock**

The primary clock on an *RCIM*. It is a 64-bit non-interrupting counter that counts each tick of the clock (400 nanoseconds). When connected to other RCIMs, the synchronized tick clock provides a time base that is consistent for all connected single board computers.

**Read**

A menu item and *push button* that read an existing file.

**record**

See *trace event*.

**region**

The period of time between the *mark* and the *current time*.

**release**

To let go of the currently-pressed *mouse button*.

**Reset**

A *push button* that cancels (undoes) all unapplied changes.

**Restore**

A *push button* that cancels all changes since the *dialog box* was displayed.

**Ruler**

A scrollable *display object* that appears as a hash-marked timeline within a *Column*. The Ruler may also contain reverse video “L”s indicating *lost trace events* and user-defined *marks*.

**running process box**

A *Data Box* that shows the process that is executing at the *current time line* on the associated CPU. If the *RCIM* module is used to timestamp events, this Data Box will show the process that is executing at the *current time line* on both the associated CPU and *node*.

**Save**

A menu item and *push button* that overwrite an existing *configuration file* with the current *display page*.

**Save As**

A menu item that saves the current *display page* in a new *configuration file*.

**Save Text**

A menu item that overwrites an existing summary text file with text from the *summary display area*.

**Save Text As**

A menu item that saves the current summary text from the *summary display area* into a new summary text file.

**SBC**

Single-board computer.

**scroll bar**

The narrow, rectangular graphic device used to change a display that would not otherwise fit in the *window*. It consists of a *trough*, a *slider*, and arrowhead buttons. If the slider does not fill the trough, there is a gap on one or both sides.

**Search Form**

The NightTrace form that allows you to define criteria to be used to locate a *trace event* in a *trace event file* by its configured characteristics and its location in the file.

**selection**

The *display object* that you *clicked* on. Alternatively, a selection may be the region of a text field you *dragged* the *mouse* over. For menu items, *push buttons*, and *radio buttons* NightTrace indicates selection by highlighting your choice. For *display objects*, NightTrace places handles on the display object. For dragged-over text fields, NightTrace displays that text in reverse video.

**separator**

A line that groups related *window* components or menu components.

**session**

A session consists of daemon definitions, display page configurations, string tables, qualified states, qualified events, macro definitions, named tags, previously-executed searches, and previously-executed summaries. A session also includes references to saved trace data segment files, kernel trace files, and user trace files. A session can be saved to a session configuration file and reloaded in subsequent invocations of NightTrace.

**shared memory buffer**

The intermediate destination of *trace events* before **ntraceud** copies them to the *trace event file* on disk.

**slider**

The graphic part of a *scroll bar* that you move in the *trough* to change the display. This component is sometimes called a thumb.

**spin lock**

A device used to protect a resource, for example, the *shared memory buffer*.

**start function**

A *state function* that provides information about the start event of the *most recent instance of a state*. The *state* to which the start function applies is either the *qualified state* specified to the *function*, or the state being currently defined. Thus, if a qualified state is not specified, start functions are only meaningful when used in *expressions* associated within a state definition. In addition, start functions should

not be used in a recursive manner in a **Start Expression**; a start function should not be specified in a **Start Expression** that applies to the state definition containing that **Start Expression**. Conversely, an **End Expression** may include start functions that apply to the state definition containing that **End Expression**.

## state

A state is a region of time bounded by two trace events, a *start event* and an *end event*. An instance of a state is the period of time between the start event and end event, including the start and end events themselves. Additional conditions may be specified in a state definition to further constrain the state. Instances of states do not nest; that is, once a state becomes active, events that might normally satisfy the conditions for the start event are ignored until the end event is encountered. See also *qualified state*.

## state function

The class of NightTrace *functions* which provide information about *states*, including: *start functions*, *end functions*, and *multi-state functions*.

## State Graph

A scrollable *display object* that graphically displays *states* as bars and *trace events* as vertical lines in a *Column*.

## state\_summary table

A pre-defined *format table* which contains formats for statistical displays of state *matches*, state *durations*, and state time *gaps*. It determines the default state-summary output format.

## streaming

The method used by the NightTrace of sending trace data from daemons directly to the NightTrace display.

## string table

The pre-defined or user-defined structure that allows you to group related integer values together and associate each one with a corresponding static character string. You hand-edit string tables into *configuration files*. The related structure is a *format table*.

## Summarize Form

The NightTrace form that allows you to obtain *trace event* and *state* statistics, such as minimum, maximum, average, and total values of *gaps*, *durations*, and *trace event arguments*.

## summary display area

The scrolling region of the **Summarize Form** that holds textual summary statistics.

### summary function

A *function* that takes another *expression* as a parameter (except for `summary_matches()`).

### summary syscall

A system call that is a special type of *exception*. A *syscall* is made when a user program forces a trap into the operating system via a special machine instruction. A syscall is used to request a given service from the kernel. Many library routines supplied as part of the operating system make syscalls to accomplish their functions. Syscalls can be suspended and resumed.

### syscall

System call.

### syscall graph

A *State Graph* on a kernel *display page*. It displays *states* representing system calls (*syscalls*) executing on the associated CPU.

### syscall table

A pre-defined, dynamically generated *string table* in the **vectors** file. This string table contains the names of all the possible system calls (*syscalls*) that can occur on the system.

### table

See *format table* and *string table*.

### table function

A *function* that allows you to extract information from user-defined and pre-defined *string tables* and *format tables*.

### tag

A uniquely-numbered indicator on a *Ruler* that represents an individual point of interest in the trace data (either a particular time or event) and which can be identified by a name.

### task

See *Ada task*.

### task ID

A 16-bit integer chosen by the Ada run-time executive that uniquely identifies an *Ada task* within an Ada program.



**text cursor**

The blinking vertical bar in an editable text field that shows your current edit position within the field.

**thread**

A sequence of instructions and associated data that is scheduled and executed as an independent entity. Every UNIX process linked with the Threads Library contains at least one, and possibly many, threads. Threads within a process share the address space of the process.

**thread ID**

A 16-bit integer chosen by the threads library that uniquely identifies a *thread* within a given process.

**TID**

A 32-bit integer that represents a unique context to which *trace events* can be associated. The following syntax numerically specifies a TID: *raw\_PID'task\_id*, *raw\_PID'thread\_id*, or *raw\_PID'0* (if neither *Ada tasks* nor *threads* are in use). The operating system process ID (*raw PID*) is contained in the upper 16 bits and either a *thread ID*, *task ID*, or zero is contained in the lower 16 bits.

**TID table**

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and associates NightTrace thread identifiers (*TIDs*) with thread names. This table is not used in kernel tracing.

**timestamp**

The time at which a specific *trace event* was logged. This provides the means by which the chronology of the trace events logged by multiple processes can be assembled. The timestamp is obtained from the system *interval timer*, the *Time Base Register*, or the *RCIM synchronized tick clock*, depending on either the system architecture or user-specified options to **ntraceud**.

**Time Base Register**

The system timer on the Power Hawk/PowerStack systems that *NightTrace* uses to timestamp *trace events*.

**time quantum**

The fixed period of time for which the kernel allocates the CPU to a process.

**trace context**

All *trace points* are associated with a log file (established via `trace_start`) and a thread name (established via `trace_open_thread`). If two processes (or *tasks*, or *threads*) are associated with the same log file and thread name, then they are said

to have the same trace context. If they differ in log file, thread name, or both, then they have different trace contexts.

**trace event**

A user-defined point of interest in an application's source code that NightTrace represents with an integer *trace event ID*. Alternatively this may be a predefined point of interest in the kernel. Along with the trace event ID, *NightTrace* records the *timestamp* when the trace event occurred, any arguments logged with the trace event, and the logging process identifier (*PID*).

**trace event argument**

A user-defined numeric value logged by an application via a *trace event*.

**trace event file**

An **ntraceud**-created binary file that contains sequences of *trace events* and data that your application and the **ntraceud** daemon logged.

**trace event function**

The class of NightTrace *functions* that provide information about *trace events*. They operate on either the *qualified event* specified to that function or, if unspecified, the *current trace event*. Trace event functions include *multi-event functions*.

**trace event ID**

An integer that identifies a *trace event*. User trace event IDs are in the range 0–4095, inclusive. Kernel trace event IDs are in the range 4100–4300, inclusive.

**trace point**

A place of interest in the source code. In user tracing, at each trace point in your application you call a trace event logging routine to log a *trace event*, possibly with additional data describing part of your program's *state* at that time. Kernel trace points and trace events are already defined and embedded in the kernel source.

**trough**

The graphic part of a *scroll bar* that holds the *slider*.

**vector table**

A pre-defined, dynamically generated *string table* in the **vectors** file. This string table contains the *interrupt* and *exception* vector names associated with the system on which the kernel tracing was performed.

**View mode**

The *display page* mode that allows you to see, search for, and summarize *trace event* information in the *message display area*, the *summary display area*, and *display*

*objects* on the *grid*; create, edit, and configure *macros*, *qualified events*, and *qualified states*. The other display-page mode is *Edit mode*.

**widget**

A *window* component, for example a *scroll bar* or *push button*.

**window**

A rectangular screen area that permits the display and/or entry of data. The Night-Trace display utility consists of several windows.

**window manager**

The program that controls *window* placement, size, and operations.

**wraparound mode**

The mode that causes the **ntraceud** daemon to intentionally discard old events. There are two forms of wraparound mode: *buffer-wraparound* and *file-wraparound*. The other **ntraceud** mode is *expansive mode*.



## Symbols

.Xdefaults file **B-1**, B-3  
/dev A-3  
/dev/spl A-3  
/etc/conf/mtune.d/trace A-2  
/usr/bin/ntracekd 7-1  
/usr/bin/ntraceud 6-1  
/usr/include/ntrace.h 2-1, 2-25, 6-23  
/usr/lib/libntrace.a 2-25  
/usr/lib/X11/fonts B-1

## A

Ada language  
    compiling and linking 2-26  
Ada task identifier 3-44, 3-45, 10-52, 11-5, 11-23,  
    11-46, 11-64, 11-86  
Apply push button 8-4, 9-39  
arg function 11-4, **11-16**  
arg\_dbl function **11-17**  
arg1 function 4-22, 11-4, 11-110  
arg2 function 11-7  
avg function **11-99**

## B

boolean table 4-17  
Box  
    last exception 13-10  
    last interrupt 13-9  
    last syscall 13-12  
    Node 13-8  
    running process 13-8  
Buffer-wraparound mode 2-22, 6-4, 6-13, 6-17, 6-18,  
    A-1, A-4

## C

C language  
    compiling and linking 2-26  
    source considerations **2-1**  
C thread identifier 10-52  
Center push button 9-32  
**clock\_gettime(3C)** routine 2-1, C-1  
**clock\_synchronize(1M)** command 2-8  
Color display 13-13  
Column 8-2, 9-29, **10-6**, B-2  
Comments  
    configuration file 4-13  
    event-map file 4-10  
Common configuration parameters 10-50  
Configuration file **4-13**  
Configuration form 10-48  
Configuration parameters  
    common 10-50  
    Fill Style 10-45  
    Then-Expression 11-105, 11-108, 11-110  
Configuring  
    display object 10-15  
Conserving disk space 6-28, A-1, A-4  
Constant string literals 4-21, 11-8, 11-106  
Constant times 11-3  
Context switch 13-1, 13-4  
    lines 13-9, 13-10, 13-12  
**cpp(1)** command 2-25  
cpu function 10-51, **11-25**  
CPU number  
    logical 13-7  
    physical 13-7  
Create menu 10-1, 10-13  
Create mouse operation 10-12  
crossref trace\_flush\_and\_trace\_trigger 6-17  
Current Time field 9-30, 9-38  
Current time line 13-7, 13-10  
    centering 9-32  
Cutoff 6-4, 6-5, 6-18, A-3, C-1

## D

- Data Box **10-5**, **10-19**, 11-108, 13-8, 13-10, 13-11, 13-13, B-2
- Data Graph 9-29, **10-8**, **10-39**, 13-10, B-2
  - Fill Style configuration parameter 10-45
- Debugger
  - NightView 1-4, 2-1
- device table 4-19, 13-4, **13-14**
- device\_nodename table 4-19, **13-15**
- df (1M)** command 6-12, 6-14, A-2, A-3
- Directory
  - /dev A-3
  - /etc/conf/mtune.d A-2
  - /usr/lib/X11/fonts B-1
- Disabling
  - IPL usage 6-4, 6-9, 6-28, A-3
  - library routines 2-2, 2-17, **2-25**
  - page locking 6-4, 6-11, 6-28, A-3
  - trace events **2-18**, 6-4, 6-24
  - tracing 2-17, **2-25**
- Discarding trace events 2-22, A-1, A-4, C-1
- Display
  - color 13-13
- Display object 1-3, **10-1**
  - Column 8-2, 9-29, **10-6**, B-2
  - configuring 10-15
  - creating 10-12
  - Data Box **10-5**, **10-19**, 11-108, 13-8, 13-10, 13-11, 13-13, B-2
  - Data Graph 9-29, **10-8**, **10-39**, 13-10, B-2
  - Event Graph **10-6**, **10-26**
  - EventGraph 9-29, B-2
  - Grid Label **10-17**
  - GridLabel **10-4**, 13-9, B-2
  - moving 10-14
  - overlapping 10-15
  - placement 10-13
  - resizing 10-14
  - Ruler 9-32, 9-33, **10-47**, A-1, B-2
  - selecting 10-13
  - State Graph 9-29, **10-7**, **10-32**, 11-111, 13-11, 13-13, B-2
- Display object configuration parameters
  - common 10-50
  - Fill Style 10-45
  - Then-Expression 11-105, 11-108, 11-110
- Display page area
  - grid 9-29, B-2
  - interval scroll bar C-1
  - message display area 8-1, 8-4, 9-29, 9-33, 10-7, 10-8, 10-9, 10-14
- Dotted area. see Grid

## Duration

state **11-74**

## E

- Edit mode 5-18
- Enabling
  - trace events 2-18, 6-4, 6-26
- End Event field 9-30, 9-38
- End functions 11-55
- End Time field 9-30, 9-38
- end\_arg function **11-57**
- end\_arg\_dbl function **11-58**
- end\_cpu function **11-66**
- end\_id function **11-56**
- end\_lwpid function **11-62**
- end\_node\_id function **11-69**
- end\_node\_name function **11-72**
- end\_num\_args function **11-59**
- end\_offset function **11-67**
- end\_pid function **11-60**
- end\_pid\_table\_name function **11-70**
- end\_raw\_pid function **11-61**
- end\_task\_id function **11-64**
- end\_thread\_id function **11-63**
- end\_tid function **11-65**
- end\_tid\_table\_name function **11-71**
- end\_time function **11-68**
- errno 3-98
- Event
  - gap 11-35
  - matches **11-36**
  - qualified 11-113
- Event Count field 9-38
- Event Graph **10-6**, **10-26**
- Event ID. see Trace event ID
- event table **4-16**
- Event. see Trace event
- event\_arg\_dbl\_summary table **4-23**
- event\_arg\_summary table **4-23**
- event\_gap function **11-35**
- event\_matches function **11-36**
- event\_summary table **4-23**
- EventGraph 9-29, B-2
- Event-map file 2-14, 4-2, **4-10**
- Exception 13-3, 13-10, 13-14, 13-15, **13-17**
  - graph 13-10
  - reference 13-18
  - resumption **13-3**, 13-10
  - suspension **13-3**, 13-10
- exec (2)** service 2-7, 2-11

- Expansive mode **6-2**, 6-4, 6-5, 6-12, 6-13, 6-14, A-1, A-4
- Expressions
  - constant string literals 4-21, 11-8, 11-106
  - functions 11-4
  - macros 4-9, 9-1
  - operands 11-2
  - operators 11-1
  - qualified events 4-9, 4-11, 9-1
  - qualified states 4-9, 9-1
  
- F**
  
- Field
  - Current Time 9-30, 9-38
  - End Event 9-30, 9-38
  - End Time 9-30, 9-38
  - Event Count 9-38
  - Increment 9-30
  - Start Event 9-30, 9-37
  - Start Time 9-30, 9-37
  - Summary-Expression 11-110
  - Time Length 9-37
- Field editing
  - multiple fields 8-4
  - single fields 8-4
- File
  - .Xdefaults **B-1**, B-3
  - /dev/spl A-3
  - /etc/conf/mtune.d/trace A-2
  - /usr/bin/ntracekd 7-1
  - /usr/bin/ntraceud 6-1
  - /usr/include/ntrace.h 2-1, 2-25, 6-23
  - /usr/lib/libntrace.a 2-25
  - configuration **4-13**
  - event-map 2-14, 4-2, **4-10**
  - rgb.txt B-1, B-3
  - trace event 1-5, 2-5, 4-9, 6-1, 6-12, 6-13, A-4
  - vectors 4-16, 13-2, 13-14, 13-15
- File system
  - NFS A-2, A-3, C-2
- File-wraparound mode 6-4, 6-5, 6-12, A-1, A-4
- Fill Style configuration parameter 10-45
- Finding. *see* Searching
- Flushing shared memory buffer **2-21**, 6-5, **6-13**, 6-18, 6-21, 6-28, A-1, A-2
- Fonts B-1
- fork(2)** service 2-7
- Form
  - Configuration 10-48
  - Search C-1
- Format
  - functions 11-104
  - format function **11-110**, 11-111
- Format table 4-9, **4-19**, 11-108
  - event\_arg\_dbl\_summary **4-23**
  - event\_arg\_summary **4-23**
  - event\_summary **4-23**
  - get\_format function 4-23, **11-108**, 11-111
  - state\_summary **4-23**
- Fortran language
  - compiling and linking 2-26
- Functions 11-4
  - arg 11-4, **11-16**
  - arg\_dbl **11-17**
  - arg1 4-22, 11-4, 11-110
  - arg2 11-7
  - avg **11-99**
  - cpu 10-51, **11-25**
  - end 11-55
  - end\_arg **11-57**
  - end\_arg\_dbl **11-58**
  - end\_cpu **11-66**
  - end\_id **11-56**
  - end\_lwpid **11-62**
  - end\_node\_id **11-69**
  - end\_node\_name **11-72**
  - end\_num\_args **11-59**
  - end\_offset **11-67**
  - end\_pid **11-60**
  - end\_pid\_table\_name **11-70**
  - end\_raw\_pid **11-61**
  - end\_task\_id **11-64**
  - end\_thread\_id **11-63**
  - end\_tid **11-65**
  - end\_tid\_table\_name **11-71**
  - end\_time **11-68**
  - event\_gap **11-35**
  - event\_matches **11-36**
  - format 11-104
  - format **11-110**, 11-111
  - get\_format 4-23, **11-108**, 11-111
  - get\_item **11-106**
  - get\_string 4-19, 4-21, 4-22, **11-104**
  - id 10-51, **11-15**, 11-108, 11-110
  - lwpid **11-21**
  - max **11-98**
  - max\_offset **11-102**
  - min **11-97**
  - min\_offset **11-101**
  - multi-event 11-35
  - multi-state 11-73
  - node\_id **11-28**
  - node\_name **11-31**
  - num\_args **11-18**
  - offset 11-77

offset 4-22, **11-26**  
offset\_arg **11-79**  
offset\_arg\_dbl **11-80**  
offset\_cpu **11-88**  
offset\_id **11-78**, 11-101, 11-102  
offset\_lwpid **11-84**  
offset\_node\_id **11-90**  
offset\_node\_name **11-93**  
offset\_num\_args **11-81**  
offset\_pid **11-82**  
offset\_pid\_table\_name **11-91**  
offset\_process\_name **11-94**  
offset\_raw\_pid **11-83**  
offset\_task\_id **11-86**  
offset\_task\_name **11-95**  
offset\_thread\_id **11-85**  
offset\_thread\_name **11-96**  
offset\_tid **11-87**  
offset\_tid\_table\_name **11-92**  
offset\_time **11-89**  
pid **11-19**, 11-108  
pid\_table\_name **11-29**  
process\_name **11-32**  
raw\_pid **11-20**  
start 11-37  
start\_arg **11-39**  
start\_arg\_dbl **11-40**  
start\_cpu **11-48**  
start\_id 11-4, **11-38**  
start\_lwpid **11-44**  
start\_node\_id **11-51**  
start\_node\_name **11-54**  
start\_num\_args **11-41**  
start\_offset **11-49**  
start\_pid **11-42**  
start\_pid\_table\_name **11-52**  
start\_raw\_pid **11-43**  
start\_task\_id **11-46**  
start\_thread\_id **11-45**  
start\_tid **11-47**  
start\_tid\_table\_name **11-53**  
start\_time **11-50**  
state\_dur **11-74**  
state\_gap 11-4, **11-73**  
state\_matches **11-75**  
state\_status **11-76**  
sum **11-100**  
summary 11-97  
summary\_matches **11-103**  
table 11-104  
task\_id **11-23**  
task\_name **11-33**  
thread\_id **11-22**  
thread\_name **11-34**

tid **11-24**  
tid\_table\_name **11-30**  
time **11-27**  
trace event 11-14

## G

### Gap

event 11-35  
state 11-73  
get\_format function 4-23, **11-108**, 11-111  
get\_item function **11-106**  
get\_string function 4-19, 4-21, 4-22, **11-104**  
Global process identifier 3-35, 3-36, 10-1, 10-51, 10-52,  
11-5, 11-19, 13-2

### Graph

data 9-29, **10-8**, 13-10  
event 9-29, **10-6**  
exception 13-10  
interrupt 13-9  
state 9-29, **10-7**, 11-111, 13-11, 13-13  
syscall 13-12

### Graphical user interface B-1

resources 13-13  
Grid 9-29, B-2  
Grid Label **10-17**  
GridLabel **10-4**, 13-9, B-2  
GridObject B-3

## H

Hardclock interrupts 13-10, 13-17

### Help

ntraceud 6-7

## I

**iconnect(3C)** routine 13-16  
id function 10-51, **11-15**, 11-108, 11-110  
**idtune(1M)** command 6-14, 6-16, A-2  
Increment 9-23  
Increment field 9-30  
Inter-process communication 2-4  
Interrupt 13-1, 13-2, 13-9, 13-14, 13-15, **13-15**, 13-16  
device-related 13-16  
graph 13-9  
hardclock 13-10, 13-17  
non-device-related 13-16



user-level 6-9, 6-11  
 Interval 1-6  
   scroll bar C-1  
 IPL register  
   disabling 6-9  
   failure to attach 2-8  
   use 6-9

## K

Kernel  
   buffer allotment A-2  
 Kernel tracing 1-1, 4-16, 4-17, 13-1

## L

Language  
   Ada 2-26  
   C **2-1**, 2-26  
   Fortran 2-26  
 Last exception box 13-10  
 Last interrupt box 13-9  
 Last syscall box 13-12  
 libntrace.a 2-25  
 Library routines 2-1  
   disabling 2-2  
   overloading in Ada 2-2  
   return values 2-2  
   trace\_begin **2-5**, 2-11, 2-16, 2-19, 2-24, 6-1,  
     6-28, C-1  
   trace\_close\_thread **2-23**  
   trace\_disable **2-17**, 6-24  
   trace\_disable\_all **2-17**, 2-25  
   trace\_disable\_range **2-17**, 6-24  
   trace\_enable **2-17**, 6-26  
   trace\_enable\_all **2-17**  
   trace\_enable\_range **2-17**, 6-26  
   trace\_end 2-8, 2-21, **2-24**, 6-3, 6-17, 6-21  
   trace\_event **2-12**, 10-1  
   trace\_event\_arg **2-12**  
   trace\_event\_dbl **2-12**  
   trace\_eventflt **2-12**  
   trace\_event\_four\_arg **2-12**  
   trace\_event\_two\_dbl **2-12**  
   trace\_event\_twoflt **2-12**  
   trace\_flush **2-21**, 6-3, 6-5, 6-13, 6-14, 6-17,  
     6-28, A-2, A-3  
   trace\_open\_thread **2-10**, 2-16, 2-19, 2-23,  
     10-52  
   trace\_trigger **2-21**, 6-3, 6-17, A-2, A-3, A-4

Lightweight process identifier 3-40, 3-41, 10-52, 11-5,  
 11-21, 11-44, 11-62, 11-84  
 Loading  
   trace event 4-5, A-5  
 Locating. see Searching  
 Logging  
   trace event 1-3, 1-4, 6-12, 6-13, **6-24**, **6-26**, A-4,  
     C-1  
 Loss  
   trace event 2-16, 2-22, 6-16, 6-28, **A-1**, C-1  
 LWPID 3-40, 3-41, 10-52, 11-5, 11-21, 11-44, 11-62,  
 11-84  
 lwpid function **11-21**

## M

Macros 4-9, 9-1, 11-113  
 Map file. see Event-map file  
 Mark 10-47  
   push button 9-33  
 Matches  
   event **11-36**  
   state 11-75  
   summary 11-103  
 max function **11-98**  
 max\_offset function **11-102**  
 Maximum value 10-44, 11-98, 11-102  
 Memory size 6-4, 6-5, 6-16, A-3, C-1  
 Menu  
   Create 10-1, 10-13  
 Message display area 8-1, 8-4, 9-29, 10-7, 10-8, 10-9,  
 10-14  
   statistics 9-33  
 min function **11-97**  
 min\_offset function **11-101**  
 Minimum value 10-44, 11-97, 11-101  
 Mode  
   buffer-wraparound 2-22, 6-4, 6-13, 6-17, 6-18, A-1,  
     A-4  
   Edit 5-18  
   expansive **6-2**, 6-4, 6-5, 6-12, 6-13, 6-14, A-1, A-4  
   file-wraparound 6-4, 6-5, 6-12, A-1, A-4  
   View 8-1, 8-3, 9-1  
 Motif 1-6  
 Mouse button  
   1 8-2, 9-31, 9-33, 10-13  
   2 8-2, 9-29, 9-31, 9-33, 10-7, 10-8, 10-9, 10-14  
   3 8-2, 9-29, 9-33, 10-9, 10-15  
 Mouse operation  
   create 10-12  
   move 10-14  
   resize 10-14

select 10-13  
Move mouse operation 10-14  
**mpadvise (3C)** routine A-4  
Multi-event functions 11-35  
Multi-state functions 11-73

## N

name\_pid table 4-17, **13-14**  
name\_tid table 4-18  
NFS file system A-2, A-3, C-2  
NightStar tool kit 1-1  
NightTrace  
    environment defaults 6-2  
    product 1-1  
NightTrace thread identifier 3-41, 3-42, 10-1, 10-51,  
    10-52, 11-5, 11-24, 11-47, 11-65, 11-87  
NightView debugger 1-4, 2-1  
Node box 13-8  
Node identifier 11-28  
Node identifier  
    ending trace event 11-69  
    offset 11-90  
    starting trace event 11-51  
Node name 11-31  
    ending trace event 11-72  
    ordinal trace event 11-93  
    starting trace event 11-54  
node\_id function **11-28**  
node\_name function **11-31**  
node\_name table 4-18, **13-14**  
NT\_CONTINUE 2-10, 2-14, 6-16  
NT\_M\_BUFFERWRAP. see Buffer-wraparound mode  
NT\_M\_DEFAULT. see Expansive mode  
NT\_M\_FILEWRAP. see File-wraparound mode  
ntrace 1-3  
    format tables 4-9, **4-19**  
    functions 11-4  
    operands 11-2  
    operators 11-1  
    performance considerations 4-5, A-5  
    string tables 4-9, **4-15**  
    viewing strategy 8-2  
ntrace field  
    Current Time 9-30, 9-38  
    End Event 9-30, 9-38  
    End Time 9-30, 9-38  
    Event Count 9-38  
    Increment 9-30  
    Start Event 9-30, 9-37  
    Start Time 9-30, 9-37  
    Time Length 9-37

ntrace functions 11-4  
ntrace macros 4-9, 9-1  
ntrace mode  
    Edit 5-18  
    View 8-1, 8-3, 9-1  
ntrace option  
    --end (load events before constraint) 4-4  
    --end (load events before constraint) A-5  
    --filestats (list statistics and trace events) A-1  
    --listing (list trace events) 4-12  
    --nohardclock (strip hardclock) A-5  
    --process (load process's events) A-5  
    --start (load events after constraint) 4-3  
    --start (load events after constraint) A-5  
ntrace qualified events 4-9, 4-11, 9-1  
ntrace qualified states 4-9, 9-1, 11-8, 11-38, 11-39,  
    11-40, 11-41, 11-42, 11-43, 11-44, 11-45,  
    11-46, 11-47, 11-48, 11-49, 11-50, 11-51,  
    11-52, 11-53, 11-54, 11-56, 11-57, 11-58,  
    11-59, 11-60, 11-61, 11-62, 11-63, 11-64,  
    11-65, 11-66, 11-67, 11-68, 11-69, 11-70,  
    11-71, 11-72, 11-73, 11-74, 11-75, 11-76  
ntrace window  
    Configuration 10-48  
    Global A-1  
    Search C-1  
ntrace.h 2-1, 2-25, 6-23  
ntracekd  
    daemon **7-1**  
ntraceud  
    buffer-full cutoff. see ntraceud  
    cutoff  
    cutoff 6-4, 6-5, 6-18, A-3, C-1  
    daemon 1-3, **6-1**  
    flush mechanism 6-4  
    help 6-7  
    invoking 6-28  
    memory size 6-4, 6-5, 6-16, A-3, C-1  
    page-fault handling 6-4  
    performance considerations 6-1, 6-17, 6-18, **A-1**  
    quit running 6-21, 6-28, 6-29  
    reset 6-20  
    shared memory buffer size. see ntraceud  
    memory size  
    sleep interval 6-3, 6-4, 6-17  
    statistical information 6-22, A-1  
    timeout interval 6-4, 6-5, 6-17, A-3  
    trace event file size 6-4, 6-12  
    trace event logging 6-4  
    version information 6-8  
ntraceud mode  
    buffer-wraparound 2-22, 6-4, 6-13, 6-17, 6-18, A-1,  
    A-4

expansive **6-2**, 6-4, 6-5, 6-12, 6-13, 6-14, A-1, A-4  
 file-wraparound 6-4, 6-5, 6-12, A-1, A-4  
 ntraceud option  
 -bufferwrap (buffer-wraparound mode) 6-4, 6-13,  
 6-17, 6-18, A-1, A-4  
 -cutoff (cutoff percentage) 6-4, 6-5, 6-18, A-3, C-1  
 -disable (disable logging) 6-4, 6-24  
 -enable (enable logging) 6-4, 6-26  
 -filewrap (file-wraparound mode) 6-4, 6-5, 6-12,  
 A-1, A-4  
 -help (help) 6-7  
 -ipldisable (do not set IPL) 6-4, 6-9, 6-28, A-3  
 -lockdisable (do not lock pages) 6-4, 6-11, 6-28,  
 A-3  
 -memsize (memory size) 6-4, 6-5, 6-16, A-3, C-1  
 -quit (quit running) 6-21, 6-29  
 -reset (reset ntraceud) 6-20  
 -stats (statistical information) 6-22, A-1  
 -timeout (timeout interval) 6-4, 6-5, 6-17, A-3  
 -version (version information) 6-8  
 num\_args function **11-18**

## O

Object. see Display object  
 Offset 4-3, 8-5, **9-29**, 9-37, 10-1, 11-4, 11-7, 11-8,  
 11-77, 11-78, 11-79, 11-80, 11-81, 11-82,  
 11-83, 11-84, 11-85, 11-86, 11-87, 11-88,  
 11-89, 11-90, 11-91, 11-92, 11-93, 11-94,  
 11-95, 11-96  
 offset function 4-22, **11-26**  
 Offset functions 11-77  
 offset\_arg function **11-79**  
 offset\_arg\_dbl function **11-80**  
 offset\_cpu function **11-88**  
 offset\_id function **11-78**, 11-101, 11-102  
 offset\_lwpid function **11-84**  
 offset\_node\_id function **11-90**  
 offset\_node\_name function **11-93**  
 offset\_num\_args function **11-81**  
 offset\_pid function **11-82**  
 offset\_pid\_table\_name function **11-91**  
 offset\_process\_name function **11-94**  
 offset\_raw\_pid function **11-83**  
 offset\_task\_id function **11-86**  
 offset\_task\_name function **11-95**  
 offset\_thread\_id function **11-85**  
 offset\_thread\_name function **11-96**  
 offset\_tid function **11-87**  
 offset\_tid\_table\_name function **11-92**  
 offset\_time function **11-89**  
 Operands

constants 11-2  
 functions 11-4  
 macros 4-9, 9-1  
 qualified events 4-9, 4-11, 9-1  
 qualified states 4-9, 9-1, 11-8, 11-38, 11-39, 11-40,  
 11-41, 11-42, 11-43, 11-44, 11-45, 11-46,  
 11-47, 11-48, 11-49, 11-50, 11-51, 11-52,  
 11-53, 11-54, 11-56, 11-57, 11-58, 11-59,  
 11-60, 11-61, 11-62, 11-63, 11-64, 11-65,  
 11-66, 11-67, 11-68, 11-69, 11-70, 11-71,  
 11-72, 11-73, 11-74, 11-75, 11-76  
 Operands in expressions 11-2  
 Operators in expressions 11-1  
 Options. see ntrace option  
 Options. see ntraceud option  
 Options. see System configuration option

## P

P\_PLOCK A-3  
 Page  
 configuration file **4-13**  
 lock disable 6-11  
 lock privilege 6-28, A-3  
 Performance considerations  
 ntrace 4-5, A-5  
 ntraceud 6-1, 6-17, 6-18, **A-1**  
 PID 3-35, 3-36, 10-1, 10-51, 10-52, 11-5, 11-19, 13-2  
 pid function **11-19**, 11-108  
 pid table **4-16**, 13-15  
 PID table name 11-29  
 pid\_nodename table 4-18, **13-14**  
 pid\_table\_name function **11-29**  
 Pre-defined tables 4-16, 4-23, 13-4, 13-13  
**printf (3S)** routine 4-12, 4-21, 11-110  
 Privilege  
 page lock 6-28, A-3  
 Process box 13-8  
 Process identifier  
 ending trace event 11-70  
 offset 11-91  
 starting trace event 11-52  
 Process identifier table name 11-29  
 Process name 11-32  
 ordinal trace event 11-94  
 process\_name function **11-32**  
 Push button  
 Apply 8-4, 9-39  
 Center 9-32  
 Mark 9-33  
 Refresh 9-36  
 Reset 9-39

Zoom In 9-24, 9-35  
Zoom Out 9-24, 9-35, C-1  
Zoom Region 9-35

## Q

Qualified events 4-9, 4-11, 9-1, 11-113  
Qualified states 4-9, 9-1, 11-8, 11-38, 11-39, 11-40, 11-41, 11-42, 11-43, 11-44, 11-45, 11-46, 11-47, 11-48, 11-49, 11-50, 11-51, 11-52, 11-53, 11-54, 11-56, 11-57, 11-58, 11-59, 11-60, 11-61, 11-62, 11-63, 11-64, 11-65, 11-66, 11-67, 11-68, 11-69, 11-70, 11-71, 11-72, 11-73, 11-74, 11-75, 11-76, 11-116

## R

Raw PID 10-52  
raw PID 3-38, 3-39, 11-5, 11-20, 11-43, 11-61, 11-83  
raw\_pid function **11-20**  
Record. see Trace event  
Refresh push button 9-36  
Reset push button 9-39  
Resize mouse operation 10-14  
Resizing  
    display objects 10-14  
Return values 2-2  
rgb.txt file B-1, B-3  
Ruler 9-32, 9-33, **10-47**, A-1, B-2  
**run (1)** command A-4  
Running process box 13-8

## S

Scroll bar C-1  
Search form C-1  
Searching  
    trace event 1-3, 1-6, 8-1, **12-1**  
Select mouse operation 10-13  
Shared memory  
    buffer 1-5, 6-13, 6-16  
    failure to attach 2-8  
    flushing **2-21**, 6-5, **6-13**, 6-18, A-1, A-2  
SHMMAX 6-14, 6-16, A-2  
Start Event field 9-30, 9-37  
Start functions 11-37  
Start Time field 9-30, 9-37  
start\_arg function **11-39**

start\_arg\_dbl function **11-40**  
start\_cpu function **11-48**  
start\_id function 11-4, **11-38**  
start\_lwpid function **11-44**  
start\_node\_id function **11-51**  
start\_node\_name function **11-54**  
start\_num\_args function **11-41**  
start\_offset function **11-49**  
start\_pid function **11-42**  
start\_pid\_table\_name function **11-52**  
start\_raw\_pid function **11-43**  
start\_task\_id function **11-46**  
start\_thread\_id function **11-45**  
start\_tid function **11-47**  
start\_tid\_table\_name function **11-53**  
start\_time function **11-50**  
State 1-2, 2-15, **10-1**, 10-7, 10-32, 13-9, 13-10  
    duration **11-74**  
    gap 11-73  
    matches 11-75  
    qualified 11-116  
State Graph 9-29, **10-7**, **10-32**, 11-111, 13-11, 13-13, B-2  
state\_dur function **11-74**  
state\_gap function 11-4, **11-73**  
state\_matches function **11-75**  
state\_status function **11-76**  
state\_summary table **4-23**  
Statistics  
    multi-event 11-35  
    multi-state 11-73  
    ntrace 9-33, A-1  
    ntraceud 6-22, A-1  
    summary 11-97  
String table 4-9, **4-15**, 11-104, 11-106  
    boolean 4-17  
    device 4-19, 13-4, **13-14**  
    device\_nodename 4-19, **13-15**  
    event **4-16**  
    get\_item function **11-106**  
    get\_string function 4-19, 4-21, 4-22, **11-104**  
    name\_pid 4-17, **13-14**  
    name\_tid 4-18  
    node\_name 4-18, **13-14**  
    pid **4-16**, 13-15  
    pid\_nodename 4-18, **13-14**  
    syscall 4-19, 13-4, **13-14**, 13-18  
    syscall\_nodename 4-19, **13-15**  
    tid **4-17**  
    tid\_nodename 4-19  
    vector 4-19, 13-2, 13-3, **13-14**, 13-15  
    vector\_nodename 4-19, **13-15**  
sum function **11-100**  
Summarize form fields

- Summary-Expression 11-110
  - Summarizing
    - trace event 1-3, **12-12**
    - trace session 1-6
  - Summary
    - matches 11-103
  - Summary functions 11-97
  - summary\_matches function **11-103**
  - Summary-Expression field 11-110
  - Syscall 13-4, 13-12, 13-14
    - graph 13-12
    - resumption 13-4
    - suspension 13-4, 13-12
  - syscall table 4-19, 13-4, **13-14**, 13-18
  - syscall\_nodename table 4-19, **13-15**
  - System call 13-4, 13-12, 13-14
- T**
- Table
    - boolean 4-17
    - device 4-19, 13-4, **13-14**
    - device\_nodename 4-19, **13-15**
    - event **4-16**
    - event\_arg\_dbl\_summary **4-23**
    - event\_arg\_summary **4-23**
    - event\_summary **4-23**
    - format 4-9, 4-19, 11-108
    - functions 11-104
    - name\_pid 4-17, **13-14**
    - name\_tid 4-18
    - node\_name 4-18, **13-14**
    - pid **4-16**, 13-15
    - pid\_nodename 4-18, **13-14**
    - pre-defined 4-16, 4-23, 13-4, 13-13
    - state\_summary **4-23**
    - string 4-9, 4-15, 11-104, 11-106
    - syscall 4-19, 13-4, **13-14**, 13-18
    - syscall\_nodename 4-19, **13-15**
    - tid **4-17**
    - tid\_nodename 4-19
    - vector 4-19, 13-2, 13-3, **13-14**, 13-15
    - vector\_nodename 4-19, **13-15**
  - Tag 10-47
  - Task name 11-33
    - ordinal trace event 11-95
  - task\_id function **11-23**
  - task\_name function **11-33**
  - Text field
    - Current Time 9-30, 9-38
    - End Event 9-30, 9-38
    - End Time 9-30, 9-38
    - Event Count 9-38
    - Increment 9-30
    - Start Event 9-30, 9-37
    - Start Time 9-30, 9-37
    - Summary-Expression 11-110
    - Time Length 9-37
  - Then-Expression configuration parameter 11-105, 11-108, 11-110
  - Thread event
    - ordinal 11-92
  - Thread identifier
    - ending trace event 11-71
    - offset 11-92
    - starting trace event 11-53
  - Thread identifier table name 11-30
  - Thread name 11-34
    - ordinal trace event 11-96
  - Thread names 4-2, 4-17, 10-52
  - thread\_id function **11-22**
  - thread\_name function **11-34**
  - TID 3-41, 3-42, 10-1, 10-51, 10-52, 11-5, 11-24, 11-47, 11-65, 11-87
  - tid function **11-24**
  - tid table **4-17**
  - TID table name 11-30
  - tid\_nodename table 4-19
  - tid\_table\_name function **11-30**
  - time function **11-27**
  - Time Length field 9-37
  - Timeout interval 6-4, 6-5, 6-17, A-3
  - Times
    - constant 11-3
  - Timestamp 1-2, **4-2**, 9-30, 11-27, 11-50, 11-68, 11-89
  - tr\_activate() 3-93
  - tr\_append\_table() 3-102
  - tr\_arg\_dbl() 3-34
  - tr\_arg\_dbl\_() 3-35
  - tr\_arg\_int() 3-33
  - tr\_arg\_int\_() 3-33
  - TR\_BUFFER\_COUNT tunable parameter A-2, C-2
  - tr\_cancel\_cb() 3-104
  - tr\_cb\_t 3-3
  - tr\_close() 3-19
  - tr\_cond\_and() 3-77
  - tr\_cond\_cb() 3-105
  - tr\_cond\_cb\_func\_t 3-4
  - tr\_cond\_copy() 3-78
  - tr\_cond\_cpu() 3-59
  - tr\_cond\_cpu\_clear() 3-59
  - tr\_cond\_create() 3-54
  - tr\_cond\_expr\_and() 3-73
  - tr\_cond\_expr\_or() 3-74
  - tr\_cond\_find() 3-55
  - tr\_cond\_func\_and() 3-70

tr\_cond\_func\_clear() 3-72  
tr\_cond\_func\_or() 3-68  
tr\_cond\_func\_t 3-4  
tr\_cond\_id() 3-56  
tr\_cond\_id\_clear() 3-58  
tr\_cond\_id\_range() 3-57  
tr\_cond\_name() 3-79  
tr\_cond\_node() 3-66  
tr\_cond\_node\_clear() 3-67  
tr\_cond\_not() 3-75  
tr\_cond\_offset() 3-82  
tr\_cond\_or() 3-76  
tr\_cond\_pid() 3-60  
tr\_cond\_pid\_clear() 3-62  
tr\_cond\_pid\_name() 3-61  
tr\_cond\_register() 3-81  
tr\_cond\_reset() 3-55  
tr\_cond\_satisfy() 3-79  
tr\_cond\_satisfy\_() 3-80  
tr\_cond\_t 3-5  
tr\_cond\_tid() 3-63  
tr\_cond\_tid\_clear() 3-65  
tr\_cond\_tid\_name() 3-64  
tr\_copy\_input() 3-98  
tr\_cpu() 3-45  
tr\_cpu\_() 3-46  
tr\_create\_table() 3-101  
tr\_destroy() 3-13  
tr\_dir\_t 3-5  
TR\_EOF 3-5, 3-23, 3-24, 3-25, 3-26, 3-82, 3-94, 3-95  
tr\_error\_check() 3-16  
tr\_error\_clear() 3-15  
TR\_EXCEPTION\_ENTRY trace event 13-3  
TR\_EXCEPTION\_EXIT trace event 13-3  
TR\_EXCEPTION\_RESUME trace event 13-3  
TR\_EXCEPTION\_SUSPEND trace event 13-3  
tr\_get\_item() 3-100  
tr\_get\_string() 3-99  
tr\_halt() 3-104  
tr\_id() 3-29  
tr\_id\_() 3-29  
tr\_init() 3-13  
TR\_INTERRUPT\_ENTRY trace event 13-2  
TR\_INTERRUPT\_EXIT trace event 13-3  
tr\_iterate() 3-103  
tr\_lwpid() 3-40  
tr\_lwpid\_() 3-41  
tr\_nargs() 3-31  
tr\_nargs\_() 3-32  
tr\_next\_event() 3-23  
tr\_next\_event\_() 3-24  
TR\_NO\_CB 3-105, 3-106  
TR\_NO\_COND 3-54, 3-55, 3-75, 3-76, 3-77, 3-78  
TR\_NO\_HANDLE 3-13  
TR\_NO\_STATE 3-84, 3-85  
tr\_node() 3-47  
tr\_node\_() 3-48  
tr\_offset\_t 3-5  
tr\_open\_file() 3-17  
tr\_open\_stream() 3-18  
TR\_PAGEFLT\_ADDR trace event 13-5, 13-11  
tr\_pid() 3-35  
tr\_pid\_() 3-36  
tr\_prev\_event() 3-24  
tr\_prev\_event\_() 3-25  
tr\_process\_name() 3-48  
tr\_process\_name\_() 3-49  
TR\_PROTFLT\_ADDR trace event 13-5, 13-11  
tr\_raw\_pid() 3-38  
tr\_raw\_pid\_() 3-39  
tr\_search() 3-26  
tr\_seek() 3-27  
tr\_state\_action\_t 3-6  
tr\_state\_active() 3-96  
tr\_state\_active\_() 3-97  
tr\_state\_cb() 3-106  
tr\_state\_cb\_func\_t 3-6  
tr\_state\_create() 3-84  
tr\_state\_end\_cond() 3-92  
tr\_state\_end\_cond\_clear() 3-92  
tr\_state\_end\_id() 3-88  
tr\_state\_end\_id\_clear() 3-90  
tr\_state\_end\_id\_range() 3-89  
tr\_state\_find() 3-85  
tr\_state\_info() 3-94  
tr\_state\_info\_() 3-95  
tr\_state\_info\_t 3-7  
tr\_state\_name() 3-85  
tr\_state\_start\_cond() 3-90  
tr\_state\_start\_cond\_clear() 3-91  
tr\_state\_start\_id() 3-86  
tr\_state\_start\_id\_clear() 3-88  
tr\_state\_start\_id\_range() 3-87  
tr\_state\_t 3-7  
tr\_stream\_consume() 3-21  
tr\_stream\_event\_t 3-8  
tr\_stream\_func\_t 3-8  
tr\_stream\_notify() 3-20  
TR\_STREAM\_SAVE 3-18  
tr\_stream\_size() 3-21  
tr\_string\_node 3-8  
TR\_SWITCHIN trace event 13-2  
TR\_SYSCALL\_ENTRY trace event 13-4  
TR\_SYSCALL\_EXIT trace event 13-4  
TR\_SYSCALL\_RESUME trace event 13-4  
TR\_SYSCALL\_SUSPEND trace event 13-4  
tr\_t 3-8  
tr\_task\_id() 3-44

- tr\_task\_id() 3-45
- tr\_task\_name() 3-50
- tr\_task\_name\_() 3-50
- tr\_thread\_id() 3-43
- tr\_thread\_id\_() 3-43
- tr\_thread\_name() 3-51
- tr\_thread\_name\_() 3-51
- tr\_tid() 3-41
- tr\_tid\_() 3-42
- tr\_time() 3-30
- tr\_time\_() 3-31
- Trace event 1-2, 10-1
  - arguments **2-14**, 4-2, 4-11, 4-14, 10-1, 10-3, 10-9, 11-16, 11-17, 11-18, 11-39, 11-40, 11-41, 11-57, 11-58, 11-59, 11-79, 11-80, 11-81
  - average size 6-16
  - context switch 13-2
  - disabling **2-18**, 6-4, 6-24
  - discarding 2-22, A-1, A-4, C-1
  - enabling 2-18, 6-4, 6-26
  - exception 13-3
  - file 1-5, 2-5, 4-9, 6-1
  - file size 6-12, 6-13, A-4
  - functions 11-14
  - ID 1-2, **2-14**, 2-18, 4-2, 4-9, 4-11, 6-24, 6-26, 10-50, C-1
  - information 10-7, 10-8, 11-14
  - interrupt 13-2
  - loading 4-5, A-5
  - logging 1-3, 1-4, 6-12, 6-13, **6-24**, **6-26**, A-4, C-1
  - loss 2-16, 2-22, 6-16, 6-28, **A-1**, C-1
  - node identifier (ending trace event) 11-69
  - node identifier (offset) 11-90
  - node identifier (starting trace event) 11-51
  - node identifier 11-28
  - node name 11-31
  - node name (ending trace event) 11-72
  - node name (ordinal trace event) 11-93
  - node name (starting trace event) 11-54
  - NT\_CONTINUE 2-10, 2-14, 6-16
  - offset 11-77
  - offset. see Offset
  - ordinal 11-90, 11-91, 11-93, 11-94, 11-95, 11-96
  - ordinal number. see Offset
  - PID table name 11-29
  - process identifier (ending trace event) 11-70
  - process identifier (offset) 11-91
  - process identifier (starting trace event) 11-52
  - process identifier table name 11-29
  - process name 11-32
  - process name (ordinal trace event) 11-94
  - searching 1-3, 1-6, 8-1, **12-1**
  - summarizing 1-3, **12-12**
  - syscall 13-4
- task name 11-33
- task name (ordinal trace event) 11-95
- thread identifier (ending trace event) 11-71
- thread identifier (offset) 11-92
- thread identifier (starting trace event) 11-53
- thread identifier table name 11-30
- thread name 11-34
- thread name (ordinal trace event) 11-96
- TID table name 11-30
- timestamp 1-2, **4-2**, 11-27, 11-50, 11-68, 11-89
- timing distortion 2-22, 6-17
- TR\_EXCEPTION\_ENTRY 13-3
- TR\_EXCEPTION\_EXIT 13-3
- TR\_EXCEPTION\_RESUME 13-3
- TR\_EXCEPTION\_SUSPEND 13-3
- TR\_INTERRUPT\_ENTRY 13-2
- TR\_INTERRUPT\_EXIT 13-3
- TR\_PAGEFLT\_ADDR 13-5, 13-11
- TR\_PROTFLT\_ADDR 13-5, 13-11
- TR\_SWITCHIN 13-2
- TR\_SYSCALL\_ENTRY 13-4
- TR\_SYSCALL\_EXIT 13-4
- TR\_SYSCALL\_RESUME 13-4
- TR\_SYSCALL\_SUSPEND 13-4
- Trace event. see Event
- Trace point 1-2, 1-4, 2-14
- trace\_begin **2-5**, 2-11, 2-16, 2-19, 2-24, 6-1, 6-28, C-1
- trace\_close\_thread **2-23**
- trace\_disable **2-17**, 6-24
- trace\_disable\_all **2-17**, 2-25
- trace\_disable\_range **2-17**, 6-24
- trace\_enable **2-17**, 6-26
- trace\_enable\_all **2-17**
- trace\_enable\_range **2-17**, 6-26
- trace\_end 2-8, 2-21, **2-24**, 6-3, 6-17, 6-21
- trace\_event **2-12**, 10-1
- trace\_event\_arg **2-12**
- trace\_event\_dbl **2-12**
- trace\_event\_flt **2-12**
- trace\_event\_four\_arg **2-12**
- trace\_event\_two\_flt **2-12**
- trace\_flush **2-21**, 6-3, 6-5, 6-13, 6-14, 6-17, 6-28, A-2, A-3
- trace\_open\_thread **2-10**, 2-16, 2-19, 2-23, 10-52
- trace\_trigger **2-21**, 6-3, 6-17, A-2, A-3, A-4
- Tracing
  - disabling 2-17, **2-25**
  - kernel 1-1, 4-16, 4-17, 13-1
  - user 1-1

## **U**

**umask(1)** command 6-1  
User tracing 1-1  
User-level interrupts 6-9, 6-11

## **V**

vector table 4-19, 13-2, 13-3, **13-14**, 13-15  
vector\_nodename table 4-19, **13-15**  
vectors file 4-16, 13-2, 13-14, 13-15  
Version  
    ntraceud 6-8  
View mode 8-1, 8-3, 9-1  
Viewing strategy  
    ntrace 8-2

## **W**

Window  
    Configuration 10-48  
    Global A-1  
    Search C-1

## **X**

X Window System  
    desk accessories 1-6  
    resources 1-6, 13-13  
**xrdb(1)** command B-1  
**xterm(1)** utility 1-6

## **Z**

Zoom In push button 9-24, 9-35  
Zoom Out push button 9-24, 9-35, C-1  
Zoom Region push button 9-35







**Spine for 1.5" Binder**

**Product Name: 0.5" from  
top of spine, Helvetica,  
36 pt, Bold**

**Volume Number (if any):  
Helvetica, 24 pt, Bold**

**Volume Name (if any):  
Helvetica, 18 pt, Bold**

**Manual Title(s):  
Helvetica, 10 pt, Bold,  
centered vertically  
within space above bar,  
double space between  
each title**

**Bar: 1" x 1/8" beginning  
1/4" in from either side**

**Part Number: Helvetica,  
6 pt, centered, 1/8" up**

**NightTrace**

**Manual**

**0890398**

