NightTrace User's Guide

Version 7.1.1

(RedHawk™ Linux®)

concurrent

0898398-164
January 2009

Preface

Scope of Manual

This manual is a reference document and user's guide for NightTrace™ - a graphical, interactive debugging and performance analysis tool.

Structure of Manual

The manual includes four major parts as shown below:

- Event Logging and Capture – Chapters 2 through 6
- Graphical Analysis – Chapters 7 through 17
- Programmatic Analysis – Chapter 18
- Reference – appendices and index

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:

italic

Books, reference cards, and items that the user must specify appear in *italic* type. Special terms and comments in code may also appear in *italic*.

`list bold`

User input appears in **`list bold`** type and must be entered exactly as shown. Names of directories, files, commands, options and man page references also appear in **`list bold`** 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.

window

Keyboard sequences and window features such as push buttons, radio buttons, menu items, labels, and 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.

Contents

Chapter 1 Introduction

User Trace Point Placement 1-2
Kernel Trace Point Placement 1-2
Timestamps 1-3
Languages 1-3
Information Displayed 1-4

Chapter 2 Using the NightTrace Logging API

Language-Specific Source Considerations 2-1
C 2-1
Fortran 2-2
Ada 2-2
Java 2-3
Inter-Process Communication and Library Routines 2-5
Understanding NightTrace Library Calls 2-6
trace_begin, Trace.begin 2-7
trace_event, Trace.event and their variants 2-13
trace_enable, trace_disable, and their variants 2-20
trace_flush, Trace.flush, trace_trigger, and Trace.trigger 2-24
trace_set_thread_name, Trace.setThreadName 2-26
trace_close_thread, Trace.closeThread 2-27
trace_end, Trace.end 2-29
trace_diag_mode 2-31
trace_diag_func 2-32
Disabling Tracing 2-33
Threads and Logging 2-33
Compiling and Linking 2-34
C Compilation and Linking 2-34
Fortran Compilation and Linking 2-34
Ada Example 2-34
Java Example 2-35

Chapter 3 Capturing User Events with ntraceud

The ntraceud Daemon 3-1
ntraceud Modes 3-2
The Default User Daemon Configuration 3-2
ntraceud Options 3-3
Invoking ntraceud 3-6

Chapter 4 Capturing Kernel Events with ntracekd

The ntracekd Daemon 4-1

ntracekd Modes 4-1
ntracekd Options 4-2
ntracekd Invocations 4-5

Chapter 5 Application Illumination

Overview 5-2
Illuminator 5-2
NightLight 5-2
Work Flow Illustration 5-2
Provided Illuminators 5-3
Detail Levels 5-3
The NightLight Graphical User Interface 5-5
File 5-6
View 5-9
Tools 5-12
Help 5-13
Wizard 5-15
Navigation Panel 5-15
Common Buttons 5-17
Select Programs with Debug Information 5-18
Define an Illuminator for each Program 5-20
Select Predefined Illuminators for each Program 5-24
Relink Illuminated Programs 5-26
Activate Illuminators in each Program 5-29
Run Scripts to Launch Programs and NightTrace 5-32
Session Manager 5-37
Select Code with Debug Information 5-39
Context Menus 5-39
Building Object 5-42
Create, Customize, and Build Illuminators 5-43
Context Menu 5-43
Context Menu on Individual Illuminators 5-45
Relink Programs 5-47
Context Menus 5-47
Path 5-48
Relink Command 5-48
Illuminators 5-50
Activation Sets 5-52
Settings For "main" Illuminator 5-53
Settings For Ordinary Illuminators 5-54
Context Menu for an Illuminator 5-54
Context Menu for a Program 5-55
Context Menu for an Activation Set 5-56
Scripts 5-58
Console 5-63
Predefined Illuminators 5-64
Detail Levels 5-64
main 5-64
glibc 5-64
pthread 5-65
ccur_rt 5-65
Illuminator Files 5-66

config.xml 5-66
next_event.txt 5-66
illuminator.**h** 5-66
illuminator.**map** 5-66
illuminator_level.**fmt** 5-66
illuminator_level.**o** 5-66
illuminator_level.**list** 5-67
illuminator.**o** 5-67
illuminator.**vararg** 5-67
NightLight Command Line Mode 5-68
Commands for Manipulating an Illuminator 5-68
nlight --create 5-68
nlight --populate 5-71
nlight --build 5-72
nlight --report 5-72
Commands for Linking with Illuminators 5-73
nlight --gcc 5-73
nlight --g77 5-74
nlight --cf77 5-74
nlight --ada 5-74
Command for Activating and Deactivating Illuminators 5-74
program 5-74
! 5-74
main[,*options]* 5-74
illuminator 5-75
level 5-75
Using NightTrace with Illuminators 5-75
Customizing an Illuminator with the Editor 5-76
Buttons 5-76
Search Editor 5-78
Options 5-80
Event IDs 5-80
Limit on Size of Aggregates Recorded 5-80
Include Functions without Dwarf Debug Info 5-80
Regular Expressions 5-81
Object Filenames 5-83
Detail Levels 5-85
Variables to Record 5-87
Groups 5-90
Create a Group 5-90
Customize a Group 5-91
Selecting Members of a Group 5-94
Functions 5-95
Adding a Function 5-95
Customizing a Function 5-96
Adding a Function to a Group 5-99
Customizing an Illuminator by Editing the config.xml File 5-101
comments 5-101
config 5-101
declare 5-102
defaults 5-102
exclude 5-103
function 5-103
group 5-104

level 5-105
caller=*{*yes|no*}* 5-106
frame=*{*yes|no*}* 5-106
aggregate_limit=*limit* 5-106
args=*{*yes|no*}* 5-106
addr_args=*{*yes|no*}* 5-107
return_val=*{*yes|no*}* 5-107
addr_ret=*{*yes|no*}* 5-107
variables=*{*yes|no*}* 5-107
errno=*{*yes|no*}* 5-107
exclude=*{*yes|no*}* 5-107
options 5-108
event_ids="*N-[M]*" 5-108
aggregate_limit="*limit*" 5-108
nodebug=*{*yes|no*}* 5-108
underscores=*{*yes|no*}* 5-108
std=*{*yes|no*}* 5-109
iregex="*regex*", xregex="*regex*" 5-109
filename="*filename*" 5-109
variable 5-109
wrapper 5-110
wrapper_file_scope 5-110
wrapper_post 5-110
wrapper_pre 5-110
wrapper_real 5-111

Chapter 6 Performance Tuning

Preventing Trace Event Loss 6-1
Daemon Scheduling Adjustment 6-1
Increasing Trace Buffer Size 6-2
Programmatic Flushing 6-3
Conserving Disk Space 6-3
Conserving Memory and Accelerating ntrace 6-3

Chapter 7 Invoking NightTrace

Command-line Options 7-1
Summary Criteria 7-6
Command-line Arguments 7-10
Trace Event Files 7-11
Event Map Files 7-11
Table Files 7-14
Tables 7-14
String Tables 7-15
Pre-Defined Strings Tables 7-17
Format Tables 7-20
Session Configuration Files 7-24
Trace Data Segments 7-25

Chapter 8 The NightTrace Main Window

Menu Bar 8-2

File 8-2
View 8-6
Daemons 8-8
Search 8-9
Summary 8-13
Profiles 8-14
Export Profiles to NightTrace API Source File 8-16
Timelines 8-18
Tools 8-26
Help 8-28
Toolbars 8-29
Pages 8-32
Panels 8-34
Preferences Dialog 8-40
Font Preferences 8-41
NightStar Global Fonts Dialog 8-45

Chapter 9 Daemons Panel

Context Menu 9-2
Control Buttons 9-6
Edit Daemon Definition 9-8
General Settings 9-9
Trace Buffer Settings 9-11
Trace Daemon Runtime Settings 9-14
Enabled Events 9-15
Triggers 9-15
Edit Triggers Dialog 9-17
Streaming Memory Usage Control 9-18
Streaming Memory Usage Control Dialog 9-19

Chapter 10 Trace Segments Panel

Trace Segments Table 10-1
Context Menu 10-2
Control Buttons 10-4

Chapter 11 Events Panel

Textual Event Tables 11-1
Context Menu 11-3

Chapter 12 Timeline Panels

Default Timeline 12-1
Current Timeline Indicator 12-2
Global Ruler 12-2
Interval Ruler 12-3
Event Graphs 12-5
Event Description Area 12-6
Keyboard Traversal 12-7
Creating Timeline Objects 12-8

Event Graph 12-10
State Graph 12-11
Data Graph 12-12
Data Graph Options Dialog 12-13
Drawing and Coloring Examples 12-16
Color Selection Dialog 12-17
Standard Color Names 12-19
Interval Ruler 12-20
Global Ruler 12-20
Label 12-20
Data Box 12-20

Chapter 13 Profiles Panels

Profile Definition Panel 13-1
Control Buttons 13-8
Summarizing Statistical Information 13-10
Condition Summaries 13-10
State Summaries 13-10
Summary Scripts 13-10
Summary Script Environment Variables 13-11
Profile Status List Panel 13-12
Profile Status List Table 13-12
Context Menu 13-13

Chapter 14 Event Descriptions Panel

Chapter 15 Tags List Panel

Creating Tags 15-1
Tags List Table 15-2
Context Menu 15-2
Control Buttons 15-3

Chapter 16 Using Expressions

Overview 16-1
Operators 16-1
Operands 16-1
Constants 16-2
Functions 16-4
Function Parameters 16-9
Function Terminology 16-11
String Functions 16-16
strcmp() 16-16
strncmp() 16-17
Trace Event Functions 16-18
id() 16-20
arg() 16-21
arg_dbl() 16-22
arg_long() 16-23
arg_long_dbl() 16-24

arg_long_long() 16-25
blk_arg() 16-26
blk_arg_bits() 16-27
blk_arg_char() 16-28
blk_arg_dbl() 16-29
blk_arg_flt() 16-30
blk_arg_long() 16-31
blk_arg_long_bits() 16-32
blk_arg_long_dbl() 16-33
blk_arg_long_long() 16-34
blk_arg_long_ubits() 16-35
blk_arg_short() 16-36
blk_arg_string() 16-37
blk_arg_ubits() 16-38
blk_arg_uchar() 16-39
blk_arg_uint() 16-40
blk_arg_ulong_long() 16-41
blk_arg_ushort() 16-42
num_args() 16-43
pid() 16-44
thread_id() 16-45
task_id() 16-46
tid() 16-47
cpu() 16-48
offset() 16-49
time() 16-50
node_id() 16-51
pid_table_name() 16-52
tid_table_name() 16-53
node_name() 16-54
process_name() 16-55
task_name() 16-56
thread_name() 16-57
Multi-Event Functions 16-58
event_gap() 16-58
event_matches() 16-59
State Functions 16-60
Start Functions 16-60
start_id() 16-62
start_arg() 16-63
start_arg_dbl() 16-64
start_arg_long() 16-65
start_arg_long_dbl() 16-66
start_arg_long_long() 16-67
start_blk_arg() 16-68
start_blk_arg_bits() 16-69
start_blk_arg_char() 16-70
start_blk_arg_dbl() 16-71
start_blk_arg_flt() 16-72
start_blk_arg_long() 16-73
start_blk_arg_long_bits() 16-74
start_blk_arg_long_dbl() 16-75
start_blk_arg_long_long() 16-76
start_blk_arg_long_ubits() 16-77

start_blk_arg_short() 16-78
start_blk_arg_string() 16-79
start_blk_arg_ubits() 16-80
start_blk_arg_uchar() 16-81
start_blk_arg_uint() 16-82
start_blk_arg_ulong_long() 16-83
start_blk_arg_ushort() 16-84
start_num_args() 16-85
start_pid() 16-86
start_thread_id() 16-87
start_task_id() 16-88
start_tid() 16-89
start_cpu() 16-90
start_offset() 16-91
start_time() 16-92
start_node_id() 16-93
start_pid_table_name() 16-94
start_tid_table_name() 16-95
start_node_name() 16-96
End Functions 16-97
end_id() 16-99
end_arg() 16-100
end_arg_dbl() 16-101
end_arg_long() 16-102
end_arg_long_dbl() 16-103
end_arg_long_long() 16-104
end_blk_arg() 16-105
end_blk_arg_bits() 16-106
end_blk_arg_char() 16-107
end_blk_arg_dbl() 16-108
end_blk_arg_flt() 16-109
end_blk_arg_long() 16-110
end_blk_arg_long_bits() 16-111
end_blk_arg_long_dbl() 16-112
end_blk_arg_long_long() 16-113
end_blk_arg_long_ubits() 16-114
end_blk_arg_short() 16-115
end_blk_arg_string() 16-116
end_blk_arg_ubits() 16-117
end_blk_arg_uchar() 16-118
end_blk_arg_uint() 16-119
end_blk_arg_ulong_long() 16-120
end_blk_arg_ushort() 16-121
end_num_args() 16-122
end_pid() 16-123
end_thread_id() 16-124
end_task_id() 16-125
end_tid() 16-126
end_cpu() 16-127
end_offset() 16-128
end_time() 16-129
end_node_id() 16-130
end_pid_table_name() 16-131
end_tid_table_name() 16-132

end_node_name() 16-133
Multi-State Functions 16-134
state_gap() 16-134
state_dur() 16-135
state_matches() 16-136
state_status() 16-137
Offset Functions 16-138
offset_id() 16-140
offset_arg() 16-141
offset_arg_dbl() 16-142
offset_arg_long() 16-143
offset_arg_long_dbl() 16-144
offset_arg_long_long() 16-145
offset_blk_arg() 16-146
offset_blk_arg_bits() 16-147
offset_blk_arg_char() 16-148
offset_blk_arg_dbl() 16-149
offset_blk_arg_flt() 16-150
offset_blk_arg_long() 16-151
offset_blk_arg_long_bits() 16-152
offset_blk_arg_long_dbl() 16-153
offset_blk_arg_long_long() 16-154
offset_blk_arg_long_ubits() 16-155
offset_blk_arg_short() 16-156
offset_blk_arg_string() 16-157
offset_blk_arg_ubits() 16-158
offset_blk_arg_uchar() 16-159
offset_blk_arg_uint() 16-160
offset_blk_arg_ulong_long() 16-161
offset_blk_arg_ushort() 16-162
offset_num_args() 16-163
offset_pid() 16-164
offset_thread_id() 16-165
offset_task_id() 16-166
offset_tid() 16-167
offset_cpu() 16-168
offset_time() 16-169
offset_node_id() 16-170
offset_pid_table_name() 16-171
offset_tid_table_name() 16-172
offset_node_name() 16-173
offset_process_name() 16-174
offset_task_name() 16-175
offset_thread_name() 16-176
Summary Functions 16-177
min() 16-177
max() 16-178
avg() 16-179
sum() 16-180
min_offset() 16-181
max_offset() 16-182
summary_matches() 16-183
Format and Table Functions 16-184
get_string() 16-184

get_item() 16-186
get_format() 16-188
format() 16-190
lookup_pc() 16-191
Profile References 16-193

Chapter 17 Kernel Tracing

Primary Kernel Trace Events 17-1
Context Switch Trace Event 17-2
Interrupt Trace Events 17-2
Exception Trace Events 17-3
Syscall Trace Events 17-4
Kernel Work Events 17-5
Additional Kernel Events 17-7
Logging Custom Kernel Events 17-8
From User Programs 17-9
From Kernel Modules 17-9
Retrieving Custom Events 17-10
Viewing Kernel Trace Event Files 17-11
Kernel Timelines 17-12
Node and CPU Information 17-13
Context Switch Information 17-13
Interrupt Information 17-14
Exception Information 17-14
System Call Information 17-15
Process Information 17-16
Kernel Events 17-16
Color Information 17-17
Kernel String Tables 17-17

Chapter 18 Using the NightTrace Analysis API

NightTrace Analysis Application Programming Interface 18-1
Data Structures 18-2
tr_arg_t 18-2
tr_cb_t 18-3
tr_cond_cb_func_t 18-3
tr_cond_func_t 18-4
tr_cond_t 18-4
tr_dir_t 18-4
tr_offset_t 18-4
tr_state_action_t 18-5
tr_state_cb_func_t 18-5
tr_state_info_t 18-6
tr_state_t 18-7
tr_stream_event_t 18-7
tr_stream_func_t 18-7
tr_string_node_t 18-7
tr_t 18-8
Functions 18-9
API Initialization and Destruction 18-14
tr_init() 18-14

tr_destroy() 18-14
Error Detection, Collection, and Reporting 18-16
tr_error_clear() 18-16
tr_error_check() 18-17
Input Specification and Streaming Control 18-18
tr_open_file() 18-18
tr_open_stream() 18-19
tr_close() 18-20
tr_stream_notify() 18-21
tr_stream_read() 18-22
tr_stream_size() 18-23
tr_free() 18-24
Event Offset Positioning 18-25
tr_next_event() 18-25
tr_next_event_() 18-26
tr_prev_event() 18-26
tr_prev_event_() 18-27
tr_search() 18-28
tr_seek() 18-29
Basic Event Attribute Functions 18-30
tr_id() 18-32
tr_id_() 18-32
tr_time() 18-33
tr_time_() 18-34
tr_nargs() 18-35
tr_nargs_() 18-35
tr_arg_int() 18-36
tr_arg_int_() 18-37
tr_arg_dbl() 18-38
tr_arg_dbl_() 18-38
tr_arg_long() 18-39
tr_arg_long_() 18-40
tr_arg_long_dbl() 18-41
tr_arg_long_dbl_() 18-41
tr_arg_long_long() 18-42
tr_arg_long_long_() 18-43
tr_arg_int_() 18-44
tr_arg_dbl() 18-45
tr_arg_dbl_() 18-45
tr_arg_long() 18-46
tr_arg_long_() 18-47
tr_arg_long_dbl() 18-48
tr_arg_long_dbl_() 18-48
tr_arg_long_long() 18-49
tr_argtype() 18-50
tr_argtype_() 18-51
tr_blk_arg() 18-51
tr_blk_arg_() 18-52
tr_blk_arg_bits() 18-53
tr_blk_arg_bits_() 18-54
tr_blk_arg_char() 18-55
tr_blk_arg_char_() 18-55
tr_blk_arg_dbl() 18-56
tr_blk_arg_dbl_() 18-57

tr_blk_arg_flt() 18-58
tr_blk_arg_flt_() 18-58
tr_blk_arg_long() 18-59
tr_blk_arg_long_() 18-60
tr_blk_arg_long_bits() 18-61
tr_blk_arg_long_bits_() 18-62
tr_blk_arg_long_dbl() 18-63
tr_blk_arg_long_dbl_() 18-63
tr_blk_arg_long_long() 18-64
tr_blk_arg_long_long_() 18-65
tr_blk_arg_long_ubits() 18-66
tr_blk_arg_long_ubits_() 18-67
tr_blk_arg_short() 18-68
tr_blk_arg_short_() 18-68
tr_blk_arg_string() 18-69
tr_blk_arg_string_() 18-70
tr_blk_arg_ubits() 18-71
tr_blk_arg_ubits_() 18-72
tr_blk_arg_uchar() 18-73
tr_blk_arg_uchar_() 18-74
tr_blk_arg_ushort() 18-75
tr_blk_arg_ushort_() 18-75
tr_pid() 18-76
tr_pid_() 18-77
tr_tid() 18-78
tr_tid_() 18-78
tr_thread_id() 18-79
tr_thread_id_() 18-80
tr_task_id() 18-81
tr_task_id_() 18-81
tr_cpu() 18-82
tr_cpu_() 18-83
tr_node() 18-84
tr_node_() 18-84
tr_process_name() 18-85
tr_process_name_() 18-86
tr_task_name() 18-86
tr_task_name_() 18-87
tr_thread_name() 18-88
tr_thread_name_() 18-88
Conditions 18-90
tr_cond_create() 18-91
tr_cond_reset() 18-92
tr_cond_find() 18-92
tr_cond_id() 18-93
tr_cond_id_range() 18-94
tr_cond_id_clear() 18-95
tr_cond_cpu() 18-96
tr_cond_cpu_clear() 18-97
tr_cond_pid() 18-98
tr_cond_pid_name() 18-99
tr_cond_pid_clear() 18-100
tr_cond_tid() 18-101
tr_cond_tid_name() 18-102

tr_cond_tid_clear() 18-103
tr_cond_node() 18-104
tr_cond_node_clear() 18-105
tr_cond_func_or() 18-106
tr_cond_func_and() 18-108
tr_cond_func_clear() 18-110
tr_cond_expr_and() 18-111
tr_cond_expr_or() 18-112
tr_cond_not() 18-113
tr_cond_or() 18-114
tr_cond_and() 18-115
tr_cond_copy() 18-116
tr_cond_name() 18-118
tr_cond_satisfy() 18-118
tr_cond_satisfy_() 18-119
tr_cond_register() 18-120
tr_cond_offset() 18-121
State-oriented Interfaces 18-122
tr_state_create() 18-122
tr_state_find() 18-123
tr_state_name() 18-124
tr_state_start_id() 18-125
tr_state_start_id_range() 18-126
tr_state_start_id_clear() 18-127
tr_state_end_id() 18-127
tr_state_end_id_range() 18-128
tr_state_end_id_clear() 18-129
tr_state_start_cond() 18-130
tr_state_start_cond_clear() 18-131
tr_state_end_cond() 18-131
tr_state_end_cond_clear() 18-132
tr_activate() 18-133
tr_state_info() 18-134
tr_state_info_() 18-135
tr_state_active() 18-136
tr_state_active_() 18-137
Output Function 18-138
tr_copy_input() 18-138
tr_copy_input_range() 18-139
String Table Functions 18-140
tr_get_string() 18-140
tr_get_item() 18-141
tr_create_table() 18-142
tr_append_table() 18-143
Callback Interfaces 18-145
tr_iterate() 18-145
tr_halt() 18-146
tr_cancel_cb() 18-146
tr_cond_cb() 18-147
tr_state_cb() 18-148

Appendix A NightStar Licensing

License Keys A-1
License Requests A-2
License Server A-2
License Reports A-3
Firewall Configuration for Floating Licenses A-3
License Support A-4

Appendix B Kernel Dependencies

Advantages for NightView. B-1
Advantages for NightTrace B-1
Advantages for NightProbe B-2
Advantages for NightTune. B-2
Frequency Based Scheduler. B-3
PCI Bar File System. B-3

Appendix C Privileged Access

Capabilities. C-1

Appendix D NightTrace Logging API Examples

Single Threaded C Example D-1
Multi-Threaded C++ Example. D-3
Fortran Example. D-5
Simple Java Example. D-6
Multi-Threaded Java Example. D-7
Rare Occurrence Example D-10

Appendix E NightTrace Analysis API Examples

list. E-2
    list.c. E-2
search E-4
    search.c E-4
watchdog E-6
    watchdog.c E-6
ptime. E-9
    ptime.c. E-10
browse E-12
    browse.c E-12
detect E-23
    detect.c E-24

Appendix F Answers to Common Questions

Appendix G Glossary

Index

Illustrations

Figure 2-1. Inter-Process Communication and Library Routines. 2-6
Figure 5-1. NightLight Main WIndow 5-5
Figure 5-2. File Menu 5-6
Figure 5-3. View Menu 5-9
Figure 5-4. Console and Editor in Tabbed Pages with Search Bar Displayed .5-11
Figure 5-5. Tools Menu 5-12
Figure 5-6. Help Menu 5-13
Figure 5-7. Wizard Navigation Panel 5-15
Figure 5-8. Wizard Common Buttons 5-17
Figure 5-9. Select Programs with Debug Information Dialog 5-18
Figure 5-10. Select Programs Advanced Settings 5-19
Figure 5-11. Define an Illuminator for each Program Dialog 5-20
Figure 5-12. Define Illuminators Advanced Menu 5-22
Figure 5-13. Define Illuminators Advanced Settings 5-23
Figure 5-14. Select Predefined Illuminators for each Program Dialog 5-24
Figure 5-15. Select Illuminators Advanced Settings 5-25
Figure 5-16. Relink Illuminated Programs Dialog 5-26
Figure 5-17. Relink Programs Advanced Settings 5-28
Figure 5-18. Activate Illuminators in each Program Dialog 5-29
Figure 5-19. Notice That Additional Illuminators Are Linked In 5-30
Figure 5-20. Activation Sets Advanced Settings 5-31
Figure 5-21. Run Scripts to Launch Programs and NightTrace Dialog in File Mode5-32
Figure 5-22. Run Scripts to Launch Programs and NightTrace Dialog in Stream Mode 5-33
Figure 5-23. Terminal Session Menu 5-35
Figure 5-24. Run Scripts Advanced Settings 5-36
Figure 5-25. Tool Tips in the Session Manager 5-37
Figure 5-26. Application Illumination Context Menu 5-38
Figure 5-27. Build Code with Debug Information Context Menu 5-39
Figure 5-28. Various Objects Added to the Session Manager 5-40
Figure 5-29. Context Menu on an Object 5-40
Figure 5-30. Build Command Dialog 5-41
Figure 5-31. Debug Info File Context Menu 5-42
Figure 5-32. Create, Customize, and Build Illuminators Context Menu 5-43
Figure 5-33. New Illuminator from Object Dialog 5-44
Figure 5-34. Context Menu on an Individual Illuminator 5-45
Figure 5-35. Relink Programs Context Menu 5-47
Figure 5-36. Individual Relinked Program Context Menu 5-48
Figure 5-37. Editing Relink Command In Place 5-48
Figure 5-38. Edit Relink Command Dialog 5-49
Figure 5-39. Relink Command Context Menu 5-49
Figure 5-40. Illuminators Context Menu 5-50
Figure 5-41. Select Illuminators Dialog 5-51

Figure 5-42. Relinked Illuminator Context Menu 5-51
Figure 5-43. Creating New Activation Set 5-52
Figure 5-44. Default Options on Illuminators 5-52
Figure 5-45. Query Current Activations Results 5-53
Figure 5-46. Context Menu on an Illuminator in an Activation Set 5-54
Figure 5-47. Context Menu on a Program in an Activation Set 5-55
Figure 5-48. Select Illuminators Dialog 5-55
Figure 5-49. Context Menu on an Activation Set 5-56
Figure 5-50. Default Activation Set in Bold 5-57
Figure 5-51. Scripts Context Menu 5-58
Figure 5-52. Run Script in Terminal Session 5-58
Figure 5-53. Invoking a Script on the Console While Passing an Option 5-59
Figure 5-54. Edit Script Dialog 5-59
Figure 5-55. Default Script created for an Activation Set (NightTrace in File Mode) 5-60
Figure 5-56. Activation Set for an Elaborate Script Example 5-61
Figure 5-57. Script for Invoking NightTrace in Stream Mode 5-61
Figure 5-58. Script Launching Instrumented Programs 5-62
Figure 5-59. Modified Script Launching Instrumented Programs 5-62
Figure 5-60. Non-zero Status Warning 5-63
Figure 5-61. Console Output 5-63
Figure 5-62. Editor Page 5-76
Figure 5-63. The Search Bar 5-78
Figure 5-64. Options 5-80
Figure 5-65. Regular Expressions Context Menu 5-81
Figure 5-66. Regular Expression Context Menu 5-83
Figure 5-67. Object Filenames Context Menu 5-83
Figure 5-68. Object Filename Context Menu 5-84
Figure 5-69. Detail Levels 5-85
Figure 5-70. Variables to Record 5-88
Figure 5-71. Add Variable Dialog 5-88
Figure 5-72. Select Variables to Record Dialog 5-89
Figure 5-73. Variable Name Context Menu 5-89
Figure 5-74. Groups Context Menu 5-90
Figure 5-75. Group Name Context Menu 5-91
Figure 5-76. Select Detail Level to Customize Dialog 5-92
Figure 5-77. A Customized Custom Detail Level for a Group 5-93
Figure 5-78. A Group with Additional Variables to Record 5-93
Figure 5-79. Member Functions Context Menu 5-94
Figure 5-80. Select Functions Dialog 5-94
Figure 5-81. Functions Context Menu 5-95
Figure 5-82. Function Context Menu 5-96
Figure 5-83. Remove Exclusion Context Menu Item 5-97
Figure 5-84. Edit Handcoded Dialog 5-98
Figure 5-85. Edit Declaration Dialog 5-99
Figure 5-86. Member Groups Context Menu 5-99
Figure 5-87. Select Groups Dialog 5-100
Figure 8-1. NightTrace Main Window 8-1
Figure 8-2. File Menu 8-2
Figure 8-3. View Menu 8-6
Figure 8-4. Toolbars Menu 8-7
Figure 8-5. Daemons Menu 8-8
Figure 8-6. Search Menu 8-9
Figure 8-7. Change Interval Dialog 8-11

Figure 8-8. Summary Menu 8-13
Figure 8-9. Profiles Menu 8-14
Figure 8-10. Export Profiles Dialog 8-16
Figure 8-11. Timelines Menu 8-18
Figure 8-12. Default User Timeline 8-19
Figure 8-13. Create Custom Kernel Timeline Dialog 8-21
Figure 8-14. Zoom sub-menu of Timelines Menu 8-22
Figure 8-15. Tools Menu 8-26
Figure 8-16. Help Menu 8-28
Figure 8-17. Tab Context Menu 8-32
Figure 8-18. Rename Page Dialog 8-32
Figure 8-19. Move Page Dialog 8-33
Figure 8-20. Page with Profile Panels 8-34
Figure 8-21. Panel Detaches from Page 8-35
Figure 8-22. Panel Movement in Progress 8-36
Figure 8-23. Profile Status List Panel on Top of Profile Definition Panel 8-37
Figure 8-24. Event Descriptions Panel added to Page 8-38
Figure 8-25. Panel in Motion Creating Tab 8-39
Figure 8-26. Font Preferences Page 8-42
Figure 8-27. NightStar Global Fonts Dialog 8-45
Figure 9-1. Daemons Panel 9-1
Figure 9-2. Daemons Panel Context Menu 9-2
Figure 9-3. Import Daemon Definitions Dialog 9-3
Figure 9-4. Attach to Running Daemons Dialog 9-4
Figure 9-5. Edit Daemon Definition Dialog 9-8
Figure 9-6. Edit Triggers Dialog 9-17
Figure 9-7. Add Triggers Entry Dialog 9-18
Figure 9-8. Streaming Memory Usage Control Dialog 9-19
Figure 10-1. Trace Segments Panel 10-1
Figure 10-2. Trace Segment Panel Context Menu 10-2
Figure 10-3. Trace Data Segment Properties Description Dialog 10-3
Figure 11-1. Events Panel 11-1
Figure 11-2. Events Panel Context Menu 11-3
Figure 11-3. Search Events for Text Dialog 11-4
Figure 11-4. Edit Event Description Dialog 11-6
Figure 12-1. Default User Timeline 12-1
Figure 12-2. Global Ruler 12-2
Figure 12-3. Interval Ruler 12-3
Figure 12-4. Event Graph with Labels 12-5
Figure 12-5. Event Description Area 12-6
Figure 12-6. Timeline Editing 12-8
Figure 12-7. Timeline Context Menu 12-9
Figure 12-8. Edit Event Graph Profile Dialog 12-10
Figure 12-9. Edit State Graph Profile Dialog 12-11
Figure 12-10. Edit Data Graph Profile Dialog 12-12
Figure 12-11. Data Graph Options Dialog 12-13
Figure 12-12. Data Graph Options Dialog Color Mode Selector 12-14
Figure 12-13. Data Graph Examples 12-17
Figure 12-14. Color Selection Dialog 12-18
Figure 12-15. Edit Data Box Profile 12-21
Figure 13-1. Profile Definition Panel 13-2
Figure 13-2. Profile Status List Panel 13-12
Figure 13-3. Profile Status List Panel Context Menu 13-13
Figure 14-1. Event Descriptions Panel 14-1

Figure 14-2. Event Description Dialog 14-2
Figure 15-1. Tags List Panel 15-1
Figure 15-2. Tags List Panel Context Menu 15-3
Figure 16-1. Function Terminology Illustrated 16-12
Figure 16-2. States and Events 16-12
Figure 17-1. Sample Kernel timeline 17-12
Figure 17-2. Node and CPU Box 17-13
Figure 17-3. Context Switch Lines 17-13
Figure 17-4. Interrupt Box and Interrupt Graph 17-14
Figure 17-5. Exception Box and Exception Graph 17-14
Figure 17-6. System Call Box and System Call Graph. 17-15
Figure 17-7. Process Information Row 17-16
Figure 17-8. Kernel Events Row 17-16
Figure 17-9. Color Key 17-17
Figure C-1. Automatically Generated Data Display Page D-5

Tables

Table 3-1. NightTrace Configuration Defaults 3-3
Table 5-1. Values Recorded As Arguments to Illumination Events 5-64
Table 5-2. Detail Levels Settings Defaults 5-85
Table 5-3. Character Entities. 5-102
Table 5-4. System Defaults 5-106
Table 12-1. Timeline Keyboard Traversal 12-7
Table 12-1. Standard Color Names 12-19
Table 16-1. Time Units and Constant Suffixes 16-3
Table 16-1. NightTrace Functions. 16-5
Table 17-1. PROCESS Event Codes. 17-6
Table 17-2. NETWORK Kernel Event Sub-ID Codes 17-6
Table 17-3. MEMORY Kernel Event Sub-ID Codes 17-7
Table C-1. Recommended /etc/pam.d Configuration C-2

1
Introduction

NightTrace is a member of the NightStar™ family of tools. NightTrace provides 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, including event occurrences, timings, and data values. NightTrace consists of the following parts:

`ntrace`

a graphical tool that controls daemon sessions and presents user and kernel trace events for interactive analysis

`ntraceud`

a daemon program that copies user applications' trace events from shared memory to trace event files

`ntracekd`

a daemon program that copies operating system kernel trace events from kernel memory to trace event files

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

`illuminator`

a command line tool for generating code to log trace events at subroutine entry and return points

`illuminate`

a command line tool for turning on and off the code generated by the **`illuminator`** tool

NightTrace operates in conjunction with other members of the NightStar RT family. NightView, a multi-process and multi-thread application debugger, provides for dynamic insertion of trace points in programs being debugged. The NightProbe data recording utility allows sampled data to be passed directly to NightTrace for graphic or textual display.

NightTrace uses the NightStar License Manager (NSLM) to control access to the

NightStar tools. See "NightStar Licensing" on page A-1 for more information.

IMPORTANT

Kernel tracing is only supported on some operating system distributions. See "Kernel Dependencies" on page B-1 for more information.

User Trace Point Placement

A *user trace point* is a place of interest in application source code. At each user trace point, you make your application log some user-specified information. This logged information is collectively called a *trace event*. Each trace event has a user-defined *trace event ID* number and optional user-supplied arguments.

Some typical user trace-point locations include:

- Suspected bug locations
- Process, subprogram, or loop entry and exit points
- Timing points
- Synchronization points for multi-process interaction
- Endpoints of atomic operations

The Application Illumination facility can be used to automatically generate user trace points for function entry and return. These trace events can include return address, parameter values, return values, etc. as arguments.

In addition to the user-supplied information, trace events automatically contain information identifying the process ID of the program generating the trace event. For multi-threaded applications, the thread ID of the specific thread generating the trace is recorded.

Kernel Trace Point Placement

Operating system distributions which support NightTtrace kernel tracing build their trace and debug kernels with trace points inserted at various points throughout the kernel source code. These trace point provide information relating to:

- System call entry and exit
- Interrupt entry and exit
- Exception entry and exit
- Kernel service routines

- Process creation, termination, and signalling
- Network activity

Analysis of kernel trace events can provide significant insight into the operation of the system and interactions between user applications. In addition to graphical displays, NightTrace provides textual description of kernel trace events which reveal useful information even for those not familiar with kernel programming.

For kernel programmers, additional custom trace events can be logged with simple kernel utility routines which can be inserted into the kernel source or in kernel module source routines.

Timestamps

Each trace event is tagged with a timestamp with sub-microsecond precision. This allows you to view and comprehend complex interactions between multiple processes and the operating system, executing on single or multiple CPU systems.

By default, an architecture-specific timing source is utilized. For Intel and AMD64, the Intel Time Stamp Counter (TSC register) is used.

If your operating system supports the Real-Time Clock and Interrupt Module (RCIM), that clock can be also used as a timestamp source.

The RCIM is a hardware module available from Concurrent Computer Corpration which provides a variety of clocks and interrupt 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.

Languages

The application programming interface for logging trace events is provided in C and Fortran for use with the following compilers:

- Concurrent Ada
- GNU C/C++
- GNU Fortran
- Intel C/C++
- Intel Fortran

- Concurrent Fortran 77

The application programming interface for trace event analysis is provided solely in C for use with C and C++ programs.

Information Displayed

The **ntrace** display utility lets you examine 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.

With the **ntrace** display utility, you can perform customized searches and summaries for individual events or user-defined states. Summaries can be generated via command line invocation of **ntrace** for generating automated reports.

2
Using the NightTrace Logging API

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

Sample programs using these functions are also provided (see “NightTrace Logging API Examples” on page D-1).

Language-Specific Source Considerations

NightTrace applications can be written in C, C++, Ada, Fortran, or Java.

The NightTrace Logging API has been tested with the following compilers:

- Concurrent Ada (MAXAda)
- Concurrent Fortran 77
- GNU C/C++
- GNU Fortran
- Intel C/C++
- Intel Fortran
- Sun Java 1.5 or later
- Aonix Perc Ultra Java 5.1 or later

For your applications to trace events, you must edit your source code and insert NightTrace library routine calls. This is called *instrumenting your code*. (The Application Illumination facility (see “Application Illumination” on page 5-1) can also be used to instrument your code without making any source changes.) 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 or 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
- Macros (described in "Disabling Tracing" on page 2-33)

The library routine return values identify the type of error, if any, the NightTrace routine encountered.

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

IMPORTANT

To fully utilize the features of NightTrace with multi-threaded applications, additional considerations must be taken into account. See the description of "Threads and Logging" on page 2-33 for more information.

Minimally, a C or C++ program can log trace points using the following sequence of library routine invocations:

```
trace_begin("file",NULL); // Called once
...
trace_event(11,2)  // Log Event ID 11 with argument 2
```

Fortran

All NightTrace library routines return `INTEGERS`, but because they begin with a "t", Fortran implicitly types them as `REAL`. You must include the NightTrace-provided file **`/usr/include/ntrace_.h`** or explicitly type them as `INTEGER` so that return values are interpreted correctly.

Minimally, a Fortran program can log trace points using the following sequences of library calls:

```
call trace_begin("data",0)    (called once)
...
call trace_event(11)
```

Ada

Ada applications can access the NightTrace library routines via the Ada package `night_trace_bindings` which is included with the MAXAda product. 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 for Linux Reference Manual.*

Java

Java applications can access the NightTrace library routines via classes in the `ntrace.logging` package. Java NightTrace class files are located in **`/usr/lib;`** be sure to add this path when using the **`-classpath`** java option or `CLASSPATH` environment variable. The Java bindings are provided via the Java Native Interface (JNI). The JNI component of the NightTrace bindings is provided in **`libntrace-java.so`**, which will be automatically loaded by the Java Virtual Machine. **`libntrace-java.so`** resides in the **`/usr/lib`** directory.

The `ntrace.logging` package contains the `Trace` class, along with two nested static classes which are used by routines in the outer `Trace` class:

`Trace.Config`

Defines a configuration object, which can be specifed to the `Trace.begin()` call to define daemon logging options.

`Trace.Error`

Exception class to hold NightTrace error returns and accessor functions to describe the specific error.

Minimally, a Java program can log trace points using the following sequences of code:

```
import ntrace.logging;
...
Trace.Begin(“data”); // (called once)
...
Trace.Event(11);
```

The Java Trace Class

Unlike C, Ada and Fortran, the files associated with the Java API do not contain a header-like file which you can refer to when coding.

The relevant public portions of the Trace class and its nested classes are described in the following sections for each routine.

However, for convenience, a listing of all relevant public portions of the Trace class is shown below:

```
public class Trace {

   public static class Error extends RuntimeException {
      public enum Msg {
         NTNOERROR,
         NTNODAEMON,
         NTNOTRACEFILE,
         NTINVALID,
         NTPERMISSION,
         NTALREADY,
         NTNOSHMID,
         NTRESOURCE,
         NTINIT,
         NTLOSTDATA,
         NTPGLOCK,
         NTNOMEM,
         NTMAPCLOCK,
         NTBADVERSION,
         NTLISTEN,
         NT_THREAD_ERR,
         DEFAULT;
      }
      public final Msg getError();
   }

   public static class Config {
      public enum ClockSource { DefaultClock, RCIMTickClock; }
      public enum PageLocking { Default, Locked, Unlocked; }
      public Config();
      public PageLocking getPageLocking();
      public boolean getDaemonSettingsPreferred();
      public int getBufferLength();
      public int getNumBuffers();
      public int getSharedMemoryPermissions();
      public ClockSource getClockSource();
      public void setPageLocking(PageLocking pl);
      public void setDaemonSettingsPreferred(boolean dsp);
      public void setBufferLength(int bl);
      public void setNumBuffers(int nb);
      public void setSharedMemoryPermissions(int smp);
      public void setClockSource(ClockSource cs);
   }

   public static void begin(String file, Config config);
   public static void begin(String file);

   public static void setThreadName(String name);

   public static void event(int id);
   public static void event(int id, int arg1);
   public static void event(int id, int arg1, int arg2);
   public static void event(int id, int arg1, int arg2, int arg3);
   public static void event(int id, int arg1, int arg2, int arg3, int
arg4);
   public static void event(int id, float arg1);
   public static void event(int id, float arg1, float arg2);
   public static void event(int id, double arg1);
   public static void event(int id, double arg1, double arg2);
```

```
   public static void event(int id, long arg1);
   public static void event(int id, long arg1, long arg2);
   public static void event(int id, String arg1);
   public static void event(int id, boolean[] arg1);
   public static void event(int id, byte[] arg1);
   public static void event(int id, char[] arg1);
   public static void event(int id, short[] arg1);
   public static void event(int id, int[] arg1);
   public static void event(int id, long[] arg1);
   public static void event(int id, float[] arg1);
   public static void event(int id, double[] arg1);

   public static void disable(int id);
   public static void disable(int id_low, int id_high);
   public static void disable();
   public static void enable(int id);
   public static void enable(int id_low, int id_high);
   public static void enable();

   public static void flush();
   public static void trigger();

   public static void closeThread();

   public static void end();

   public static void enableDiagnostics(boolean on);
}
```

Error Handling

Unlike the other language interfaces, error conditions in the Java API are handled by throwing a Trace.Error object.

Objects of that class can be caught and queried for a specific enumerated reason assocated with the error.

The public members of the Error class are shown in “The Java Trace Class” on page 2-3.

The following snippet of code demonstrates how you might use this class:

```
try {
   Trace.event(5);
} catch (Trace.Error e) {
   if (e.getError() == Trace.Error.Msg.NTINIT) {
      System.out.println("Oops; forgot to start ntraceud!");
   }
}
```

Inter-Process Communication and Library Routines

Your application logs trace events to a shared memory area. A user daemon copies trace events from shared memory buffers to the trace event file or to the NightTrace graphical analysis tool. The relationship between your application and the user daemon and the sequence of library calls needed to maintain this relationship appears in the figure below.

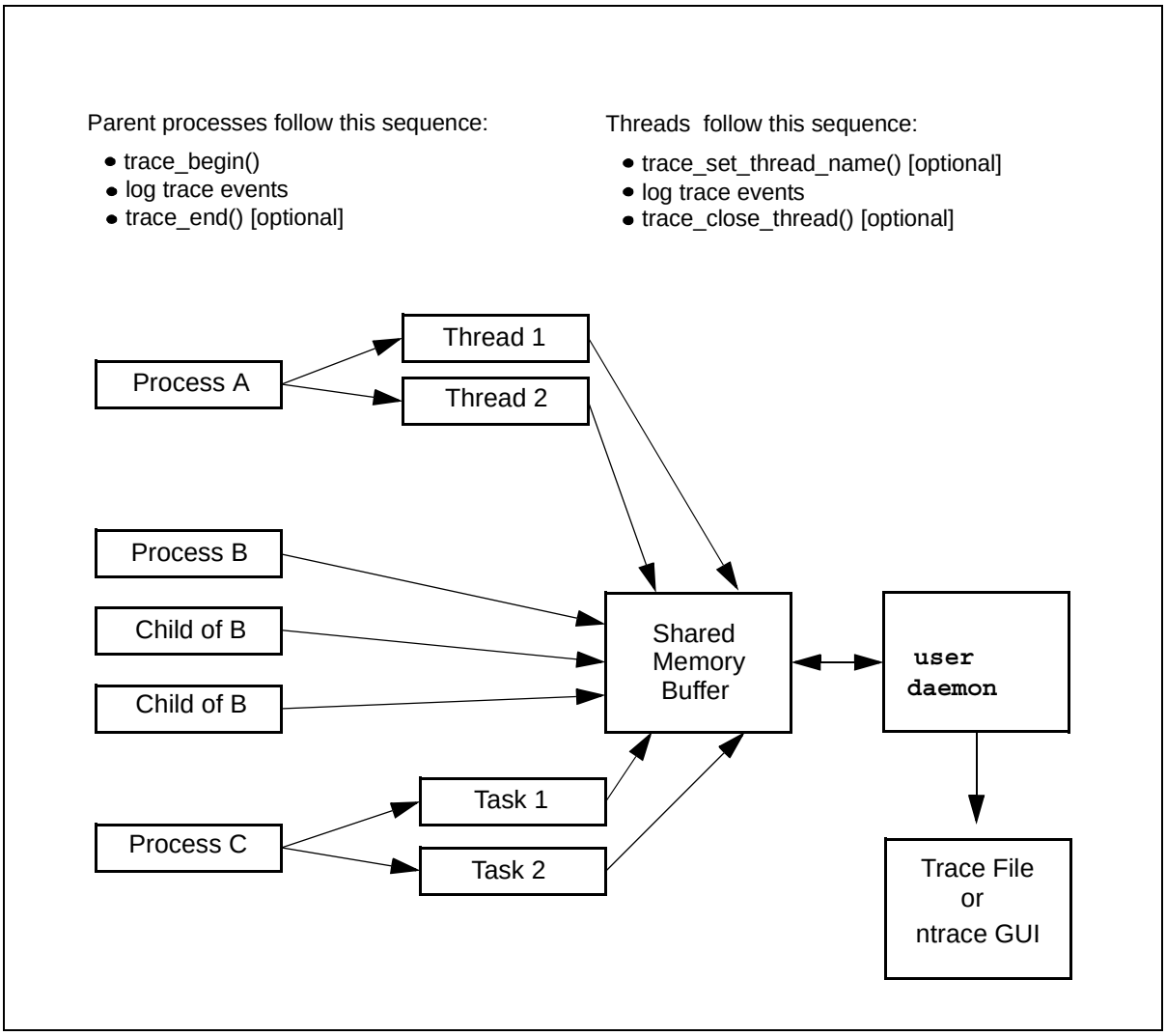


Figure 2-1. Inter-Process Communication and Library Routines

Understanding NightTrace Library Calls

There are C, Ada, Fortran, and Java versions of each NightTrace library routine. These routines perform the following functions:

- Initialize a tracing session
- Log trace events to shared memory
- Enable and disable specified trace events
- Explicitly notify the daemon to copy shared memory to disk

- Control how diagnostics are generated
- Terminate a tracing session

trace_begin, Trace.begin

The `trace_begin` and `Trace.begin` routines initialize the tracing session and acquire resources for your process.

SYNTAX

C:

```
int trace_begin(char *trace_file,
                ntconfig_t * cfg);
```

Fortran:

```
integer function trace_begin(trace_file, cfg)
character *(*) trace_file
integer cfg(NTC_SIZE)
```

Ada:

```
function trace_begin(
   trace_file : string;
   num_buffers : integer; -- default is 8
   buffer_length : integer; -- default is 32768
   lock_pages : boolean := true;
   clock : ntclock_t := NT_USE_ARCHITECTURE_CLOCK;
   shmid_perm : integer := 8#666#;
   inherit : boolean := true)
return ntrace_error;
```

Java:

```
package ntrace.logging;
public class Trace {
   static class Config {
         Config();
         enum ClockSource {
            DefaultClock,
            RCIMTickClock;}
         enum PageLocking {
            Default,
            Locked,
            Unlocked;}
         PageLocking getPageLocking();
         boolean getDaemonSettingsPreferred();
         int getBufferLength();
         int getNumBuffers();
         int getSharedMemoryPermissions()
         ClockSource getClockSource();
```

```
            void setPageLocking(PageLocking);
            void setDaemonSettingsPreferred(boolean);
            void setBufferLength(int);
            void setNumBuffers(int);
            void setSharedMemoryPermissions(int);
            void setClockSource(ClockSource);
        };
        static void begin (String trace_file);
        static void begin (String trace_file, Config cfg);
    };
```

PARAMETERS

trace_file

The user daemon logs trace events to an output file, *trace_file*. When you invoke the user daemon, you must specify this file's name. For the user daemon to log your process' trace events to this file, the trace event file parameter in your `trace_begin` call must correspond to the key file value on the daemon invocation. The names do not have to exactly match textually, but they do have to refer to the same actual pathname; for example, one path name may begin at your current working directory and the other may begin at the root directory. When a user daemon is sending trace data directly to the NightTrace graphical analysis tool, this file name serves only a handle so that the user daemon and the application can communicate -- no data is transferred to the file in this case.

cnf

C

cfg must be either a NULL pointer, in which case the default settings are used, or a pointer to a `ntconfig_t` structure.

The following function can also be used to initialize *cfg* to appropriate default values:

`void trace_default_config (ntconfig_t * config);`

Therefore, the following code sequence:

```
ntconfig_t config;
trace_default_config(&config);
trace_begin("file",&config);
```

is equivalent to:

```
trace_begin("file",NULL);
```

This is most useful when you wish to change just a few specific configuration parameters without having to explicitly define all parameters. For example:

```
ntconfig_t config;
trace_default_config(&config);
```

```
config.ntc_num_buffers = 64;
trace_begin("file",&config);
```

Ada

The individual members of the structure are supplied directly as parameters to the routine, with appropriate default values. Both the user application and the user daemon associated with it must agree on the configuration settings (or indicate that the other's settings may be preferred).

Fortran

The *cfg* record must be represented by an array of NTC_SIZE integer items. Member of the array must be provided as described below.

Java

The *cfg* parameter is optional. If specified, it must be an instance of the `Trace.Config` class. You can use the mutator methods within that class to set options in the `Trace.Config` object.

The following describe the individual parameters or mutator Java functions:

C: *ntc_version*
Fortran: *config*`(ntc_version)`
Java: n/a

The value of the `NTC_VERSION` macro from **`ntrace.h`**

C: *ntc_lock_pages*
Ada: *lock_pages*
Fortran: *cfg*`(ntc_lock_pages)`
Java: *cfg*`.setPageLocking(PageLocking)`

For C, Ada, and Fortran, one of the following values: *ntp_default*, which specifies that page locking should default; *ntp_lock*, which specifies that critical pages are to be locked in memory; or *ntp_no_lock*, which specifies that critical pages shall not be locked in memory. *ntp_default* does not request page locking, but does conflict with a user daemon configuration setting of *ntp_lock* or *ntp_no_lock*.

For Java, one of the values:

PageLocking.Default
PageLocking.Locked
PageLocking.Unlocked

which specifies that page locking should default, be locked, or be unlocked, respectively.

C: *ntc_clock*
Ada: *clock*
Fortran: *cfg*`(ntc_clock)`
Java: *cfg*`.setClockSource(ClockSource)`

Specifies which clock to use as a timing source.

For C, Ada, and Fortran, this value must be NT_USE_ARCHITECTURE_CLOCK or NT_USE_RCIM_TICK_CLOCK. The user daemon default value is NT_USE_ARCHITECTURE_CLOCK.

For Java, one of the following values:

ClockSource.Default
ClockSource.RCIMTickClock

The daemon default is to use the Default (Architecture) clock.

C: *ntc_shmid_perm*
Ada: *shmid_perm*
Fortran: *cnf*`(ntc_shmid_perm)`
Java: *cnf*`.setSharedMemoryPermissions(int)`

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: *cnf*`(ntc_daemon_preferred)`
Java: *cnf*`.setDaemonSettingsPreferred(boolean)`

When set to `TRUE`, this parameter causes conflicts between the configuration as specified by the user and by the corresponding user daemon to be resolved in favor of the daemon. Otherwise, conflicts will be resolved in favor of the first configuration that executes, which will cause the subsequent user daemon invocation or `trace_begin` (or `Trace.begin`) call to fail.

C: *ntc_num_buffers, ntc_buffer_length*
Ada: *num_buffers, buffer_length*
Fortran: *cnf*`(ntc_num_buffers)`, *cnf*`(ntc_buffer_length)`
Java: *cnf*`.setNumBuffers(int)`, *cnf*`.setBufferLength(int)`

These two parameters define the amount of memory used to hold trace events. The user daemon configuration defaults to 8 buffers which individually hold 32768 events. The values as specified here will be rounded up to the closest power of two. The units of buffer length are in units of minimally-sized events. Some trace event interfaces with additional user-specified arguments require additional space. The default daemon values for these fields are 8 buffers of length 32768.

C: *ntc_daemon_wait_usec*
Fortran: *config*`(ntc_daemon_wait_usec)`
Java: n/a

Specifies the number of microseconds the user daemon should pause between busy-wait contention for control of the shared memory buffers when flushing buffers to the output device. The user daemon configuration for this parameter defaults to 100 *us*. This value should be kept relatively short to prevent data loss if massive user application trace activity prevents the daemon from flushing the shared memory buffers.

C: *ntc_reserved*
Fortran: *cnf*`(ntc_reserved)`
Java: n/a

These parameters are reserved for future use; currently, they must be set to zero for proper future operation.

DESCRIPTION

The `trace_begin` and `Trace.begin` routines perform the following operations:

- Verify 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
- Verify the supplied configuration settings are not in conflict with a pre-existing daemon or define the configuration with these settings if the user daemon does not yet exist.
- Verify that the RCIM synchronized tick clock is counting if it was selected as the timestamp source
- Attach the shared memory buffer (after creating it if needed)
- Lock critical NightTrace library routine pages in memory as directed. Note that you must have the `CAP_SYS_NICE` capability to lock pages in memory (see "Privileged Access" on page C-1 for details).
- Initialize trace event tracing in this process

A process that results from the **`execve(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` or `Trace.begin`. Processes that result from a fork in a process that has already initialized the tracing session need not call `trace_begin`.

The `trace_begin` or `Trace.begin` routine must be called only once per parent process (unless an intervening `trace_end` or `Trace.end` call has been made).

If Application Illumination is used, the main **`illuminator`** (see "Application Illumination" on page 5-1) will perform a `trace_begin()` call. The **`illuminate`** tool (see "Command for Activating and Deactivating Illuminators" on page 5-74) can be used to set some of the parameters to this call.

RETURN CONDITIONS

C, Ada, and Fortran:

Upon successful operation, the `trace_begin` routine returns `NTNOERROR` or `NTLISTEN`; the latter in the case where no daemon has yet been started. Otherwise, an error value as defined in **`ntrace.h`** and **`ntrace_.h`** is returned, as shown in the Error Code section below.

Java:

The `Trace.begin()` routine has no return value. It returns if the call is successful (including the case of where no daemon has yet been started). Otherwise, a `Trace.Error` exception object is thrown, which further describes the error. When caught, you can use the exception object's `getError()` routine to obtain the specific error enumeration value from the `Trace.Error.Msg` enumerated type; relevant error code descriptions are shown below.

Error Code Enumerations:

`NTNOERROR`

A daemon has already been started that matches the filename passed as *key_file*.

`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.

NOTE

This error enumeration is not ever thrown by the Java API. Calls to `Trace.begin()` will silently succeed even if a matching daemon has not yet been started.

`NTALREADY`

The application has already initialized the trace without an intervening `trace_end` or `Trace.end` call. Tracing can continue in spite of this error.

`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.

`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 change the page locking configuration setting to `FALSE`. (See `ntc_lock_pages` or `Config.setPageLocking()` above).

`NTNOSHMID`

This can occur if the size of the shared memory buffer exceeds the system limits or the shared memory buffer already exists but the size required by the parameters defining the number of buffers and buffer length exceeds the current size. To increase the system limits on shared memory, adjust the *kernel.shmmni*, *kernel.shmall*, and *kernel.shmmax* parameters using **`systcl(8)`**. Use **`ipcrm(1)`** to remove the existing shared memory segment if it is not being used by another application.

SEE ALSO

- `trace_end(), Trace.end()`

trace_event, Trace.event and their 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, int arg);
int trace_event_two_arg (int ID, int arg1, int arg2);
int trace_event_three_arg (int ID, int arg1, int arg2, int arg3);
int trace_event_four_arg(int ID, int arg1, int arg2, int arg3, int
arg4);

int trace_event_long (int ID, long arg);
int trace_event_two_long (int ID, long arg1, long arg2);

int trace_event_long_long (int ID, long long arg);
```

```
int trace_event_two_long_long (int ID, long long arg1, long long
arg2);

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_long_dbl (int ID, long double arg);

int trace_event_blk(int ID, void *args, int bytes);
int trace_event_string(int ID, char *str);
```

Fortran:

```
integer function trace_event (ID)
integer ID

integer function trace_event_arg (ID, arg)
integer function trace_event_two_arg(ID, arg1, arg2)
integer function trace_event_three_arg (ID, arg1, arg2, arg3)
integer function trace_event_four_arg (ID,arg1,arg2,arg3,arg4)
integer ID, arg, arg1, arg2, arg3, arg4

integer function trace_event_long (ID, arg)
integer function trace_event_two_long (ID, arg1, arg2)
integer ID
integer arg, arg1, arg2       (32-bit OS)
integer*8 arg, arg1, arg2     (64-bit OS)

integer f unction trace_event_long_long (ID, arg)
integer function trace_event_two_long_long (ID, arg1, arg2)
integer ID
integer*8 arg, arg1, arg2

integer function trace_event_dbl (ID, arg)
integer function trace_event_two_dbl (ID, arg1, arg2)
integer ID
double precision arg , arg1, arg2
```

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;
```

Java:

```
package ntrace.logging;
class Trace {
   static void event(int ID);
```

```
    static void event(int ID, int arg);
    static void event(int ID, int arg1, int arg2);
    static void event(int ID, int arg1, int arg2, int arg3);
    static void event(int ID, int arg1, int arg2, int arg3, int arg4);
    static void event(int ID, long arg);
    static void event(int ID, long arg1, long arg2);
    static void event(int ID, double arg);
    static void event(int ID, double arg1, double arg2);
    static void event(int ID, String arg);
    static void event(int ID, char[] arg);
    static void event(int ID, int[] arg);
    static void event(int ID, double[] arg);
    static void event(int ID, byte[] arg);
    static void event(int ID, float[] arg);
    static void event(int ID, long[] arg);
    static void event(int ID, short[] arg);
    static void event(int ID, boolean[] arg);
}
```

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 Tables" on page 7-17 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. 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 7-11 for more information on event-map files and formats. Alternatively, you could call the `format` function. See "format()" on page 16-190 for details.

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.

TIP

To save time re-editing, recompiling, and relinking your application, consider beginning with many trace points in the source code. You can dynamically enable or disable specific trace events.

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 for 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.

By convention, each trace event logging invocation should log 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*. Typically, 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 2-20 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 12-11, "Event Graph" on page 12-10, and "Data Graph" on page 12-12 for more information about these display objects.

RETURN CONDITIONS

C, Ada, and Fortran:

These routines return a zero value (NTNOERROR) on successful completion. Otherwise, an error value as defined in **ntrace.h** and **ntrace_.h** is returned, as shown in the Error Code section below.

Java:

On successful completion, these routines return without any value. Otherwise, a Trace.Error exception object is thrown, which further describes the error. When caught, you can use the exception object's getError() routine to obtain the specific error enumeration value from the Trace.Error.Msg enumerated type; relevant error code descriptions are shown below.

Error Code Enumerations:

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 trace_begin or Trace.begin call precedes 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 buffers were 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 buffers can be increased using the **--num_bufs** and **--buflen** options to **ntraceud**, the User Event Buffer settings on the User Trace tab of the Daemon Definition dialog in **ntrace** tool, or the number of buffers or buffer length can be adjusted as part of thetrace_begin or Trace.begin calls.

SEE ALSO

- trace_flush(), Trace.flush()
- trace_trigger(), Trace.trigger()
- trace_enable(), Trace.enable()

- `trace_enable_range()`
- `trace_enable_all()`
- `trace_disable(), 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`, `trace_disable_all`, and `Trace.disable` routines respectively make your application ignore requests to log one or more trace events. The `trace_enable`, `trace_enable_range`, `trace_enable_all` and `Trace.enable` 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;
```

Java:

```
package ntrace.logging;
class Trace {
   static void enable(int ID);
   static void enable(int ID_low, int ID_high);
   static void enable();
   static void disable(int ID);
   static void disable(int ID_low, ID_high);
   static void disable();
}
```

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 event IDs (`0-4095`, inclusive). See “trace_event, Trace.event and their variants” on page 2-13 for more information.

ID_low

It is possible to manipulate groups of trace event IDs by specifying a range of trace event IDs. *ID_low* is the smallest trace event ID in the range.

ID_high

It is possible to manipulate groups of trace event IDs by specifying a range of trace event IDs. *ID_high* 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 buffers. The disable routines differ by how many trace events they disable. `trace_disable`, and `Trace.disable` with a single argument, disable one trace event ID. `trace_disable_range`, and `Trace.disable` with two arguments, disable a range of trace event IDs, including both range endpoints. `trace_disable_all`, and `Trace.disable` without any arguments, disable 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 buffers. The enable routines differ by how many trace events they enable. `trace_enable`, and `Trace.enable` with a single argument, enable one trace event ID. `trace_enable_range`, and `Trace.enable` with two arguments, enable a range of trace event IDs, including both range endpoints. `trace_enable_all`, and `Trace.enable` without arguments, enable 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 enable and 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 and then selectively enable the trace events of interest.

RETURN CONDITIONS

C, Ada, and Fortran:

These routines return a zero value (`NTNOERROR`) on successful completion. Otherwise, an error value as defined in **`ntrace.h`** and **`ntrace_.h`** is returned, as shown in the Error Code section below.

Java:

On successful completion, these routines return without any value. Otherwise, a `Trace.Error` exception object is thrown, which further describes the error. When caught, you can use the exception object's `getError()` routine to obtain the specific error enumeration value from the `Trace.Error.Msg` enumerated type; relevant error code descriptions are shown below.

Error Code Enumerations:

`NTINIT`

The NightTrace library routines were not initialized. Solution: Be sure a `trace_begin` or `Trace.begin` call precedes 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

- `trace_event`, `Trace.event` and its variants

trace_flush, Trace.flush, trace_trigger, and Trace.trigger

The flush and trigger routines asynchronously wake the user daemon and direct it to copy trace events from the shared memory buffers 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;
```

Java:

```
package ntrace.logging;
class Trace {
   static void flush();
   static void trigger();
}
```

DESCRIPTION

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

- At least one of the individual buffers is filled with trace events
- Your application calls `trace_flush`, `trace_trigger`, `trace_end`, `Trace.flush`, `Trace.trigger`, or `Trace.end`
- **`ntraceud`** is invoked with the **`--flush-now`** option

- The NightTrace graphical analysis tool requests a flush for immediately analysis of the latest trace events

TIP

The trigger functions work identically to the flush functions, except that the trigger functions work 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 (and therefore presumably less-vital) trace events when the shared memory buffer gets full. In buffer-wraparound mode, you must explicitly call `trace_flush`, `Trace.flush`, `trace_trigger`, 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 these functions too often or you will reduce the effectiveness of this mode. See "ntraceud Options" on page 3-3 for more information on buffer-wraparound mode.

RETURN CONDITIONS

C, Ada, and Fortran:

The `trace_flush` and `trace_trigger routines` return a zero value (`NTNOERROR`) on successful completion. Otherwise, an error value as defined in **`ntrace.h`** and **`ntrace_.h`** is returned, as shown in the Error Code section below.

Java:

On successful completion, these routines return without any value. Otherwise, a `Trace.Error` exception object is thrown, which further describes the error. When caught, you can use the exception object's `getError()` routine to obtain the specific error enumeration value from the `Trace.Error.Msg` enumerated type; relevant error code descriptions are shown below.

Error Code Enumerations:

`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

- `trace_event`, `Trace.event` and its variants

trace_set_thread_name, Trace.setThreadName

The `trace_set_thread_name` and `Trace.setThreadName` routines associate the current C thread, Ada task, or Java thread with a user-specified name. Use of this library routine is optional, as described in the Description paragraph below.

SYNTAX

C:

```
int trace_set_thread_name(const char *thread_name);
```

Fortran:

```
integer function trace_set_thread_name(thread_name)
character *(*) thread_name
```

Java:

```
package ntrace.logging;
class Trace {
   static void setThreadName(String thread_name);
}
```

PARAMETERS

thread_name

NightTrace's graphical displays and textual summary information indicate which threads logged trace events.

Naming your threads can make the displays much more readable. This function lets you associate a meaningful character string name with the current threads' more cryptic numeric ID. 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.

Thread names should be limited to alpha-numeric characters and should contain at least one non-numeric character. Names that are entirely numeric may be discarded if a more descriptive name is available (including the default thread name "main"). Some special characters are allowed, but their use is not recommended. Do not use the names "ALL" or "NONE" as they are used internally within NightTrace and may cause unexpected results .

DESCRIPTION

When using Java or when linking with the thread-aware version of the NightTrace Logging API library (**`libntrace_thr`**), the default thread name is formed directly from the thread's internal **`gettid(2)`** value.

For C and Ada programs, if not using the thread-aware version of the library, you cannot distinguish which threads logged which trace events -- all threads share the same name.

By default, the main program thread is called “main”.

Calling `trace_set_thread_name` or `Trace.setThreadName` sets the name of the calling thread to the specified name, overriding any previous name, default or otherwise, given to the thread.

Calling `trace_set_thread_name` or `Trace.setThreadName` multiple times for the same thread is not recommended, as it can cause confusion. Depending on the mode of trace event collection, some trace event may have the prior name and some may have the new name -- or, all trace events may have the name associated with the last call to `trace_set_thread_name`.

RETURN CONDITIONS

C, Ada, and Fortran:

The `trace_set_thread_name` routine returns a zero value (`NTNOERROR`) on successful completion. Otherwise, an error value as defined in **`ntrace.h`** and **`ntrace_.h`** is returned, as shown in the Error Code section below.

Java:

On successful completion, `Trace.setThreadName` returns without any value. Otherwise, a `Trace.Error` exception object is thrown, which further describes the error. When caught, you can use the exception object’s `getError()` routine to obtain the specific error enumeration value from the `Trace.Error.Msg` enumerated type; relevant error code descriptions are shown below.

Error Code Enumerations:

`NTINVALID`

An invalid thread name was specified.

SEE ALSO

- `trace_begin(), Trace.begin()`
- `trace_close_thread(), Trace.closeThread()`

trace_close_thread, Trace.closeThread

The `trace_close_thread` and `Trace.closeThread` routines inform the NightTrace Logging API library that the calling thread will no longer log trace events. These functions are only useful when you have a multi-threaded application which has been linked with the thread-aware version of the NightTrace Logging API library (**`libntrace_thr`**) or you have a multi-threaded Java program.

SYNTAX

C:

```
int trace_close_thread;
```

Fortran:

```
integer function trace_close_thread
```

Ada:

```
function trace_close_thread return
ntrace_error;
```

Java:

```
package ntrace.logging;
class Trace {
   static void closeThread();
}
```

DESCRIPTION

Use of this function is optional, but it is good practice to call this function for all threads which have logged trace events.

If you do not call `trace_close_thread` or `Trace.closeThread` and you have logged trace events from a thread other than the main program thread, then the shared memory resources associated with the NightTrace logging API session will remain attached to the process even after a call to `trace_end` or `Trace.end` is made.

RETURN CONDITIONS

C, Ada, and Fortran:

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.

Java:

On successful completion, `Trace.closeThread` returns without any value. Otherwise, a `Trace.Error` exception object is thrown, which further describes the error. When caught, you can use the exception object's `getError()` routine to obtain the specific error enumeration value from the `Trace.Error.Msg` enumerated type; relevant error code descriptions are shown below.

Error Code Enumerations:

`NTINIT`

The NightTrace library routines were not initialized by a call to `trace_begin` or `Trace.begin`.

SEE ALSO

- `trace_begin(), Trace.begin()`
- `trace_end(), Trace.end()`

trace_end, Trace.end

The `trace_end` and `Trace.end` routines free resources and terminate the trace session in your process. Use of these routines is not strictly necessary, since all tracing resources are automatically freed when the application exits. However, for applications that may continue to execute but have no need for subsequent tracing, calling these routines is appropriate.

SYNTAX

C:

```
int trace_end;
```

Fortran:

```
integer function trace_end
```

Ada:

```
function trace_end
return ntrace_error;
```

Java:

```
package ntrace.logging;
class Trace {
   static void end();
}
```

DESCRIPTION

This routine performs the following operations:

- Terminates trace event tracing in this process
- Flushes trace events from the shared memory buffer to the trace event file
- Detaches the shared memory buffer

NOTE

If you have a multi-threaded program linked with the thread-aware version of the NightTrace logging API, the shared memory will not be detached from the process if you have logged trace events from threads which have not yet called `trace_close_thread`.

- Notifies the user daemon that the current process has finished logging trace events

RETURN CONDITIONS

C, Ada, and Fortran:

The `trace_end` routine returns a zero value (`NTNOERROR`) on successful completion. Otherwise, an error value as defined in **`ntrace.h`** and **`ntrace_.h`** is returned, as shown in the Error Code section below.

Java:

On successful completion, `Trace.end` returns without any value. Otherwise, a `Trace.Error` exception object is thrown, which further describes the error. When caught, you can use the exception object's `getError()` routine to obtain the specific error enumeration value from the `Trace.Error.Msg` enumerated type; relevant error code descriptions are shown below.

Error Code Enumerations:

`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` or `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

- `trace_begin(), Trace.begin()`
- `trace_close_thread(), Trace.closeThread`

trace_diag_mode

The `trace_diag_mode` routine controls the generation of diagnostics for critical NightTrace API routines.

The NightTrace API diagnostic routine is called when critical errors occur for some NightTrace API routines if the diagnostic mode is set to TRUE (on).

SYNTAX

C:

```
void trace_diag_mode (int on);
```

Fortran:

```
external trace_diag_mode
```

Java:

```
package ntrace.logging;
class Trace {
   static void enableDiagnostics(boolean);
}
```

DESCRIPTION

These functions control whether diagnostic text is sent to `stderr` by NightTrace logging API routines when significant or critical errors are encountered. Regardless of the setting of the diagnostic mode, individual functions within the NightTrace logging API will use return values (or exceptions in the case of Java) to inform you of error conditions.

For C and Fortran, specify a zero value to turn diagnostics off, or a non-zero value to enable diagnostics.

For Java, pass `true` to enable diagnostics, and `false` to disasble them.

For C, the NightTrace API diagnostic routine may be changed via the `trace_diag_func` routine.

NOTE

Setting the **`NTRACE_SILENT`** environment variable to a non-null value will prevent diagnostics routines from being called, regardless of the diagnostic mode setting.

SEE ALSO

- `trace_diag_func()`

trace_diag_func

The `trace_diag_func` routine replaces the default NightTrace API diagnostic routine with one supplied with the function invocation.

SYNTAX

C:

```
void trace_diag_func (void(*func)(char*,int));
```

DESCRIPTION

The specified function is invoked when critical errors occur for some NightTrace API routines if the trace diagnostic mode is set to `TRUE`. If this function is not called, an internal NightTrace library routine is invoked when significant errors occur, which prints a diagnostics to `stderr`, unless the diagnostics have been turned off via `trace_diag_mode()`.

NOTE

Setting the **`NTRACE_SILENT`** environment variable to a non-null value will prevent diagnostics routines from being called, regardless of the diagnostic mode setting.

SEE ALSO

- `trace_diag_mode()`

Disabling Tracing

There are four ways to disable tracing in your application:

- For C applications that include **/usr/include/ntrace.h**, you must recompile your application with the **-DNNTRACE** 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.

- Call the `trace_disable_all` routine near the top of the source, recompile, and relink your application. (For more information about this routine, see "trace_enable, trace_disable, and their variants" on page 2-20.) If your application calls any of the enable routines, this method is not entirely effective.

- Start a user daemon with all events disabled.

- Do not start a user daemon.

  The trace library routines have been highly optimized to have minimal overhead, especially when no user daemon has been initiated.

Threads and Logging

In order to distinguish between multiple threads in a multi-threaded application, the following step must be taken:

- C applications must be linked with the thread-aware version of the NightTrace logging API by specifying the **-lntrace_thr** link option.

- Ada tasking applications automatically include the **-lntrace_thr** option when using the Ada NightTrace bindings.

- Threaded Java applications automatically include the **-lntrace_thr** library when using the `ntrace.logging.Trace` class.

If the thread-aware version of the library is not used , calls to log trace events from threads will succeed but cannot be distinguished from other threads or the main thread.

By default, when using the thread-aware version of the library, threads are named using their internal **gettid(2)** value. You can explicitly set the name of a thread to something more useful by calling `trace_set_thread_name` or `Trace.setThreadName`.

Compiling and Linking

You must link in the NightTrace library so that your application can initialize its trace mechanism and log trace events.

For single-threaded applications, specify the **/usr/lib/libntrace.a** library.

For multi-threaded applications, specify the **/usr/lib/libntrace_thr.a** library (Multi-threaded Java and Ada applications will automatically use the threaded NightTrace library).

C Compilation and Linking

Single-threaded example:

```
$ cc app.c -lntrace
```

Multi-threaded example:

```
$ cc app.c -lntrace_thr -lpthread
```

See "NightTrace Logging API Examples" on page D-1 for more demonstrative examples.

Fortran Compilation and Linking

RedHawk Linux:

```
$ cf77 app.f -lntrace
```

or

```
$ g77 app.f -lntrace
```

See "NightTrace Logging API Examples" on page D-1 for more demonstrative examples.

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 for Linux Reference Manual.*

Java Example

Ensure that a path to a valid Java development environment **`bin`** directory is in your $PATH variable.

```
$ javac -classpath /usr/lib:. app.java
```

See “NightTrace Logging API Examples” on page D-1 for more demonstrative examples.

3
Capturing User Events 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 "Edit Daemon Definition" on page 9-8.
- Use the command line tool **ntraceud**.

The interactive interface 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" on page 3-1
- "ntraceud Modes" on page 3-2
- "The Default User Daemon Configuration" on page 3-2
- "ntraceud Options" on page 3-3
- "Invoking ntraceud" on page 3-6

The ntraceud Daemon

When you start up **ntraceud**, it creates a daemon background process and then returns control to the invoking program, normally the shell. 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 the output device, usually a 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.

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**.

The daemon remains idle until one of the following conditions exist:

- One of the shared memory buffers fills

- You terminate execution of **ntraceud**
- Your application calls `trace_flush()`, `trace_trigger()`, or `trace_end()`
- A subsequent invocation of **ntraceud** explicitly requests a flush

ntraceud Modes

By default, **ntraceud** operates in an expansive mode, continually increasing the size of the output file as events are copied from the shared memory buffers to disk.

ntraceud also offers a file-wrap mode. This mode essentially places a limit on the maximum size the file can grow to. Once the limit is reached, the oldest events in the file are overwritten.

ntraceud also offers a buffer-wrap mode. In this mode, the shared memory buffers are filled without waking the daemon. When all buffers have been filled, the oldest events are overwritten with the newest ones. No disk activity occurs until **ntraceud** is terminated, or an explicit flush operation is requested, at which time, all buffers are copied to the output file.

Both file-wrap and buffer-wrap modes may be used together.

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 3-1 summarizes these options. Detailed descriptions of these options are described in the following section.

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, which for Intel and AMD Opteron based machines is the Time Stamp Counter (TSC register). On operating systems that support the Real-Time Clock and Interrupt Module (RCIM), the RCIM's clock can be used as a timestamp source by using the **--rcim** option to **ntraceud** (see "ntraceud Options" on page 3-3).

ntraceud and the NightTrace library routines optionally use page locking to prevent page faults during trace event logging.

A summary of NightTrace configuration defaults follows.

Table 3-1. NightTrace Configuration Defaults

Characteristic	Default	Modifying Option
Number of buffers	8	**--numbuf**s=*number*
Size of each buffer	32768 raw events	**--buflen=***len*
Buffer wrap mode	No wrapping	**--bufferwrap**
Trace event file size	Indefinite	**--filewrap**=*bytes*
Trace events enabled for logging	All	**--disable** =*ID* and **--enable**=*ID*
Page Locking	No Page Locking	**--lock**

ntraceud Options

ntraceud copies trace events from shared memory buffers to the output device, which is normally a file.

The **ntraceud** invocation syntax is:

ntraceud [*options*] *trace-filename*

The *trace-filename* parameter is required for all **ntraceud** invocations. When starting a daemon, it defines the shared memory identifier that the daemon and application will use to communicate. When requesting statistics for a running daemon or when stopping a daemon, it identifies the running daemon. Finally, unless run in streaming mode, the *trace-filename* defines the output file which will hold trace events as they are copied from memory.

The command-line options to **ntraceud** are:

--bufferwrap
-b

Collect events in the shared memory buffers, but do not output them to the output device until **ntraceud** is terminated or an explicit flush request occurs via an **ntraceud** invocation or from the NightTrace Logging API.

When the shared memory buffers are completely filled, the oldest trace events are overwritten by the newest events.

--buflen[**=***buflen*]
-Bl *buflen*

Sets the length of each of the shared memory buffers used by **ntraceud** to *buflen*. The value represents the number of parameterless events that can be stored in each buffer. The value *buflen* should be a power of 2 -- otherwise the

value is automatically adjusted by **ntraceud**. Use this option in conjunction with **--numbufs** to control the amount of shared memory to be used. The default value for *buflen* is 32768. Note that `trace_event_arg` API calls (and other similar interfaces which include parameters) consume more space than those without parameters.

Specifying a large value may exceed the system limitation on the maximum size of shared memory. You can adjust the system limitation by changing the *kernel.shmmax* and *kernel.shmall* variables via the **sysctl(8)** command.

--cpu=*cpu*

Causes the daemon to run on the CPUs specified by *cpu*. The *cpu* parameter must be a comma-separated list of logical CPUs or CPU ranges.

--disable=*ID*[*-ID*]
--enable=*ID*[*-ID*]
-d *ID*[*-ID*]
-e *ID*[*-ID*]

Disable or enable one trace event ID or a range of trace event IDs, as defined by *ID* or the range *ID-ID*, from being logged. Any number of these options may be specified. Upon the first invocation of **ntraceud** that creates the daemon process, the first **--enable** option disables all other trace events. When **ntraceud** is invoked subsequently to adjust status of events for the current session, **--enable** options only enable the specified trace events. By default, all trace events are enabled.

--filewrap=*bytes*
-fw *bytes*

Start the **ntraceud** daemon in file-wrap mode such that the maximum trace file size will be *bytes* bytes. A *K* or *M* suffix indicates that the size is in kilobyte or megabyte units, respectively. Once the maximum size has been reached, **ntraceud** overwrites the oldest trace events logged by the application.

--flush

This option forces a flush of all shared memory buffers that contain trace events. This is especially useful when the daemon is operating in bufferwrap mode or **ntraceud** is stream data to an application linked with the NightTrace Analysis API when the rate of events is relatively low.

--help
-h

Display a brief description of **ntraceud** options to *stdout* and exit.

--info
-i

Display summary information about a running **ntraceud** daemon. The display includes information about the number of events generated, events in the

shared memory buffers, events written to the output device and any data loss that has occurred.

Data loss usually occurs because your application is writing trace events to the shared memory buffers faster than **ntraceud** can copy them to the trace-event file. Limit data loss by increasing the **--numbufs** and **--buflen** option settings or using **--bufferwrap** and by executing **ntraceud** with urgent priority.

--join
-j

Allow the initiation of an **ntraceud** daemon even if a user application has already initiated a trace session using the specified *trace-filename* argument.

--lock
--nolock

Specify whether critical pages are to be locked in memory or should not be locked in memory. Note that you must have the CAP_IPC_LOCK capability to lock pages in memory (see "Privileged Access" on page C-1 for details).

--numbufs[**=***numbufs*]
-Bn *numbufs*

Sets the number of shared memory buffers used by **ntraceud** to *numbufs*. The value *numbufs* should be a power 2 -- the value is automatically adjusted by **ntraceud** if this is not the case. Use this option in conjunction with **--buflen** to control the amount of shared memory to be used. The default value of *numbufs* is 8.

Specifying a large value may exceed the system limitation on the maximum size of shared memory. You can adjust the system limitation by changing the *kernel.shmmax* and *kernel.shmall* variables via the **sysctl(8)** command.

--policy=*pol*

This option sets the scheduling policy under which the daemon will operate. The *pol* parameter must be *other*, *fifo*, or *rr*, indicating standard interactive, real-time first-in first-out or real-time round-robin scheduling, respectively. By default, *pol* is *other*. Use this option in conjunction with **--priority** and **--cpu** to adjust the scheduling attributes of **ntraceud**. See **sched_setscheduler(2)** for more information on scheduling policies. Note that you must have the CAP_SYS_NICE capability to set a real-time scheduling policy (see "Privileged Access" on page C-1 for details).

--priority=*prio*

This option sets the scheduling priority under which the daemon will operate. The *prio* parameter must be an integer priority value which is consistent with the range of priorities allowed by the associated scheduling class set via the **--policy** option. By default, *prio* is 0 and the scheduling policy is *other* which dictates normal interactive scheduling. See **sched_setscheduler(2)** for more information on scheduling priorities. Note that you must have the CAP_SYS_NICE capability to set a real-time scheduling priority (see "Privileged Access" on page C-1 for details)

--processor=*bias*

The *bias* parameter must be a comma-separated list of logical cpu numbers or ranges. This option restricts the daemon to only run on the specified cpu(s).

--quit
-q

After all processes associated with the **ntraceud** session defined by *trace-filename* have exited or called `trace_end`, flush all remaining events in the shared memory buffers, terminate the corresponding **ntraceud** daemon, remove the corresponding shared memory identifier, and close the file. This option causes **ntraceud** to wait for all processes to either exit or call `trace_end` before tracing is terminated, whereas the **--quit-now** option terminates the daemon without waiting.

--quit-now
-qn

Immediately flush all remaining events in the shared memory buffers, terminate the corresponding **ntraceud** daemon, remove the corresponding shared memory identifier, and close the file.

--rcim

Specify use of the RCIM synchronized tick clock as the timing source. This option is useful when simultaneously capturing data from multiple systems since the RCIM tick clock can be synchronized between systems.

This option is only available on operating systems that support the RCIM.

--stream

This option causes binary trace data to be output to *stdout*. This option is intended to provide streaming data to applications using the NightTrace Analysis API; e.g. **ntraceud --stream /tmp/key | a.out**. In this case, the *trace-filename* specified is not modified (although it will be created if it does not already exist).

--version
-v

Display the current **ntraceud** version to *stdout* and exit.

Invoking ntraceud

This section describes 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 following invocation might be used when tuning your NightTrace configuration because you lost trace events last time.

ntraceud --numbufs=16 --buflen=65536 *trace_file*

To eliminate any disk activity, or to run for long periods of time and only capture the latest data, the following invocation might be used.

ntraceud --bufferwrap *trace_file*

To conserve disk space for long runs, the following invocation might be used.

ntraceud --filewrap=*bytes trace_file*

The following invocation should be used when the user application is already running and you wish to start collecting trace data from it.

ntraceud --join *trace_file*

To obtain information on the status of an active daemon, the following invocation could be used:

ntraceud --info *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 Logging 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-now *trace_file*

To provide streaming trace data to an application written using the NightTrace Analysis API, the following information could be used:

ntraceud --stream *trace_file* | **./a.out**

Note that in the above invocation, the *trace_file* parameter serves only as a handle for communication between the daemon and the user application that is logging the events; no data is written to the file. The **--stream** option instructs that the binary data stream be redirected to *stdout*. See “NightTrace Analysis Application Programming Interface” on page 18-1 for more information.

4
Capturing Kernel Events 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 sections:

- "The ntracekd Daemon" on page 4-1
- "ntracekd Modes" on page 4-1
- "ntracekd Options" on page 4-2
- "ntracekd Invocations" on page 4-5

The ntracekd Daemon

When you initiate **ntracekd**, it creates a daemon background process and returns while that daemon process executes. Once it returns to the invoking process, usually the shell, 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 or an explicit flush request is made via a subsequent **ntracekd** invocation, 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:

--bufferwrap
-b

Collect events in kernel bufferwrap mode, delaying output to *filename* until stopped or flushed. This delays the disk activity normally involved in copying kernel buffers to the output file as they become full.

--cpu=*cpu*

Causes the daemon to run on the CPUs specified by *cpu*. The *cpu* parameter must be a comma-separated list of logical CPUs or CPU ranges.

--events=*events*
-e *events*

Set the state for the events listed in the list *events* to enabled or disabled. *Events* is a comma-separated list of event numbers or names preceded with a + (meaning enabled) or - (meaning disabled). A + or - without a number or name means enable or disable all, respectively. This option can be used after a daemon is already running to dynamically disable or enable events.

For example, to disable all events except those representing context switches, you could enter:

```
ntracekd --events=-,+schedchange
```

--flush

This option flushes all kernel buffers. It is particularly useful in conjunction with the **--stream** option when streaming binary data to a NightTrace Analysis API application.

--help
-H

Prints a description of the available options and exits.

--info
-i

This option can be specified to obtain statistics about a kernel daemon already initiated by a previous **ntracekd** command. It prints statistics to *stdout.*

--kill
-k

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.

--policy=*pol*

This option sets the scheduling policy under which the daemon will operate. The *pol* parameter must be *other, fifo,* or *rr,* indicating standard interactive, real-time first-in first-out or real-time round-robin scheduling, respectively. By default, *pol* is *other.* Use this option in conjunction with **--priority** and **--cpu** to adjust the scheduling attributes of **ntracekd**. See **sched_setscheduler(2)** for more information on scheduling policies.

--priority=*prio*

This option sets the scheduling priority under which the daemon will operate. The *prio* parameter must be an integer priority value which is consistent with the range of priorities allowed by the associated scheduling class set via the **--policy** option. By default, *prio* is 0 and the scheduling policy is *other* which dictates normal interactive scheduling. See **sched_setscheduler(2)** for more information on scheduling priorities.

--processor=*bias*

The *bias* parameter must be a comma-separated list of logical cpu numbers or ranges. This option restricts the daemon to only run on the specified cpu(s).

--quit
-q

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.

--raw
-x

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. This option is not normally used.

--rcim
-r

Use the RCIM tick clock as the timing source instead of the default timing source.

This option can only be used on operating systems that support the RCIM.

--size=*size*
-s *size*

This option specifies the maximum size of the output file. It is required when initiating a daemon unless the **--wait** or **--bufferwrap** options are used. *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.

--stream

This option causes output to be sent to *stdout* in binary form for use as input to a NightTrace Analysis API application. When this option is used, the *filename* parameter still required, but no data will be written to it. With **--stream** the *filename* serves solely as a communication handle between **ntracekd** invocations.

--verbose
-v

When this option is used in conjunction with **--info**, it includes the list of enabled events.

--wait=*seconds*
-w *seconds*

Start the daemon and begin kernel tracing for *seconds* before stopping the daemon.

--bufsize=*sz*
-Bs *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. The default size of a kernel buffer is 250000 bytes.

--numbufs=*n*
-Bn *n*

This option defines the number of kernel buffers. *n* must be an integer number. The number of kernel buffers defaults to 4.

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 tracing 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
```

To initiate a daemon to capture kernel data for five seconds and then terminate the daemon and view the data, the following sequence of commands might be used:

```
> ntracekd --wait=5 kernel-data
> ntrace kernel-data
```

5
Application Illumination

The challenge of debugging real-time programs is that problems are often time sensitive. Stepping through the program one statement at a time with a traditional debugger is little help in debugging such problems. Even the expedience of inserting `printf()` statements may introduce sufficient I/O overhead to interfere with the behavior of a real-time program. NightTrace's trace points have little overhead, but it can be tedious to insert large numbers of them into the source code.

Application Illumination is a facility to automatically generate trace points for function calls and returns. It patches them into the object code, and thus requires no source changes.

This chapter describes the Application Illumination facility and consists of the following sections:

- "Overview" on page 5-2
- "The NightLight Graphical User Interface" on page 5-5
- "Wizard" on page 5-15
- "Session Manager" on page 5-37
- "Console" on page 5-63
- "Predefined Illuminators" on page 5-64
- "Illuminator Files" on page 5-66
- "NightLight Command Line Mode" on page 5-68
- "Customizing an Illuminator with the Editor" on page 5-76
- "Customizing an Illuminator by Editing the config.xml File" on page 5-101

Overview

Illuminator

An *illuminator* is a directory that contains an object file with a set of "wrapper" routines, an event map and format tables for **`ntrace`** to use, and various other support files. Calls to the routines that are going to be traced will be diverted to their corresponding "wrapper" functions, which record the entry event, call the real function, record the return event, and then return to the original call site.

NightLight

NightLight (**`nlight`**) is the tool used to create, manipulate, and use illuminators. It can be used via command line options or in GUI mode.

Work Flow Illustration

The following transcript illustrates illuminating the code of a simple user program using NightLight command line options.

1. Build your code with debug information so that Application Illumination knows the signatures of your functions:

   ```
   $ gcc -g -c *.c
   $ gcc *.o
   ```

2. Create and build an illuminator called `a.ai` for the `a.out` program:

   ```
   $ nlight --build=a.ai a.out
   ```

3. Relink your program with the illuminator that was constructed in step 2, along with a predefined illuminator called `main` that performs the `trace_begin()` operation. At this point, although the illuminators are linked into the program, they are inert. Calls to the routines to be traced are still called directly. Illuminators may sit in your program unused and not interfering with performance at all until you need them.

   ```
   $ gcc *.o -o a.outAI `nlight --gcc main a.ai`
   ```

4. Activate the illuminators in **`a.outAI`**. Calls to the routines to be traced are now diverted to the "wrapper" functions.

   ```
   $ nlight --illuminate=a.outAI main a.ai
   ```

5. Start up a daemon to record the events, run the program, shut the daemon down, and run **`ntrace`**, which finds the trace file and illuminator support files from paths embedded in **`a.outAI`**:

```
$ ntraceud trace_file
$ a.outAI
$ ntraceud -q trace_file
$ ntrace a.outAI
```

Provided Illuminators

Illuminators are provided for some system libraries: `glibc`, `pthread`, and `ccur_rt`. Since the building of illuminators depends on DWARF debug information which is not normally in system libraries, creating custom illuminators for system libraries requires the installation of appropriate debug-info RPMs or versions of the system libraries with debug information still in them (different Linux distributions take differing approaches to this).

An illuminator for `main()` is also provided that will perform the `trace_begin()` operation for programs that aren't already using NightTrace (see "trace_begin, Trace.begin" on page 2-7).

Detail Levels

When activating an illuminator, a named detail level may be specified (the default one is called `2`). A detail level may be customized to trace a particular subset of the functions that can be traced and to log more or less information as arguments to the events. By default, illuminators have detail levels called `1`, `2`, and `3`, providing increasing amounts of detail recorded in the arguments of the events. Custom detail level names are not limited to numbers.

1. Relink the previous example to include the **`glibc`** illuminator:

   ```
   $ gcc *.o -o a.outAI `nlight --gcc main a.ai glibc`
   ```

2. Activate the **`a.ai`** illuminator specifying a higher level of detail than we used above, and **`glibc`** with a low level of detail:

   ```
   $ nlight --illuminate=a.outAI main a.ai=3 glibc=1
   ```

3. Start up a daemon to record the events, run the program, shut the daemon down, and run **`ntrace`**, which finds the trace file and illuminator support files from paths embedded in **`a.outAI`**:

   ```
   $ ntraceud tracefile
   $ a.outAI
   $ ntraceud -q tracefile
   $ ntrace a.outAI
   ```

Here is some sample output of a few events with detail level `3`:

```
 9: cpu=?? ENTER_regcomp    test_illuminator main         0.010745903
    calling regcomp(preg=0x60f120,pattern=0x60f170,cflags=9)
       *preg={
          buffer=0x0,
          allocated=0,
          used=0,
          syntax=0,
          ...}
       *pattern="^main$"
       caller=0x478f44
       frame=0x7fbfff5870

10: cpu=?? RETURN_regcomp   test_illuminator main         0.010800482
    returning from regcomp()=0
       errno=0

11: cpu=?? ENTER_strlen     test_illuminator main         0.010801628
    calling strlen(s=0x4bb374)
       *s=".*\.internal_io\.ada"
       caller=0x478f07
       frame=0x7fbfff5870

12: cpu=?? RETURN_strlen    test_illuminator main         0.010802240
    returning from strlen()=20
       errno=0
```

The NightLight Graphical User Interface

To invoke the NightLight graphical user interface, invoke **`nlight`** without any options:

```
$ nlight &
```

This will open the New Session window:

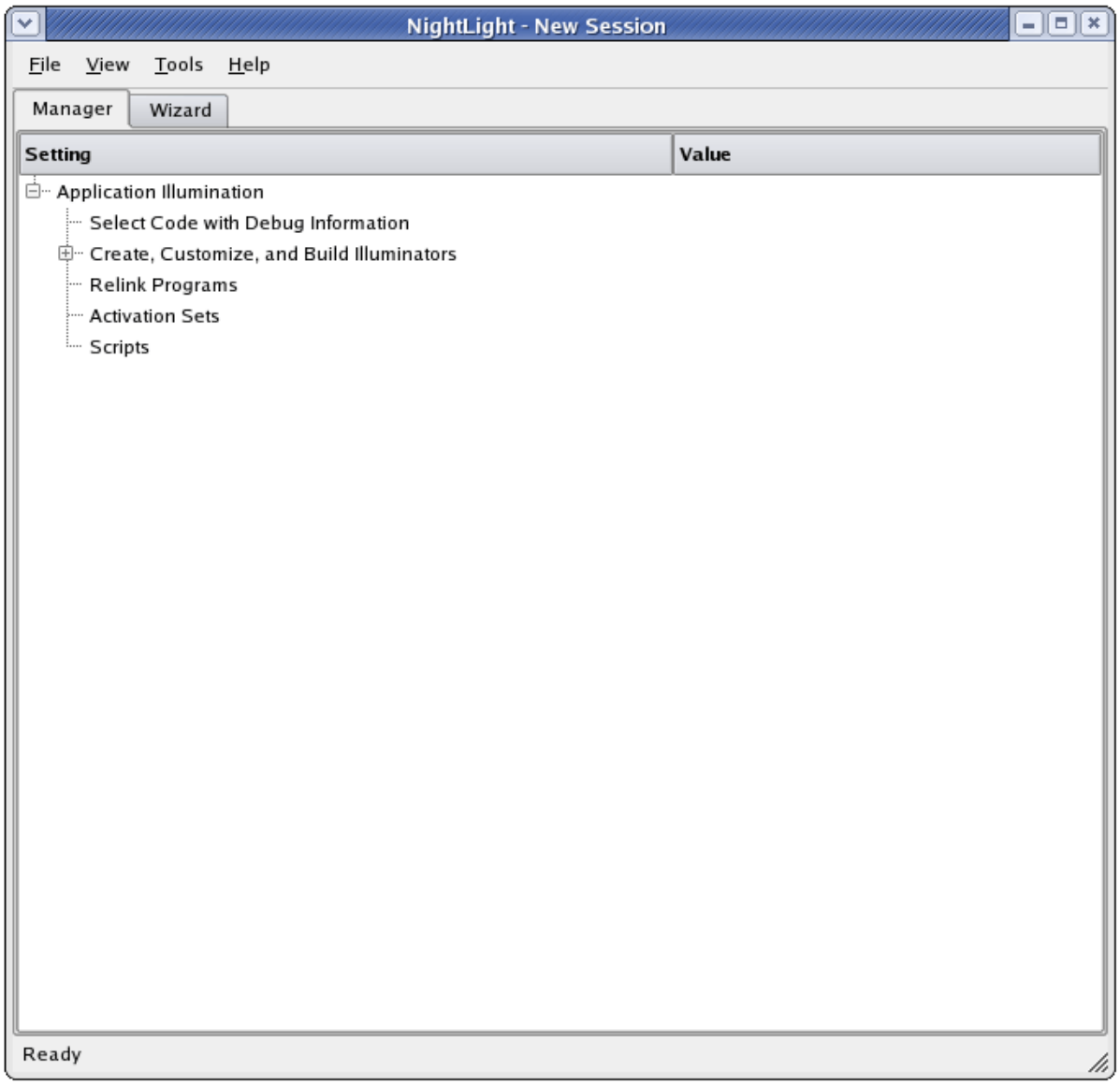


Figure 5-1. NightLight Main WIndow

You may also specify a previously saved session or an illuminator to customize on the comand line.

The five branches of the tree structure on the Manager page correspond to the five steps outlined in the "Work Flow Illustration" on page 5-2. Most actions within it are taken through context menus by right clicking on the various items in the tree. There is also a Wizard page that will guide you step-by-step through the most-used functions of the tool.

The menu bar provides access to session configuration services, additional tools, and help. The menu bar provides the following menus:

- File
- View
- Tools
- Help

Each menu is described in the sections that follow:

File

Accelerator: Alt+F

The File menu contains session-related items such as creating a new session, saving the current session or illuminator, and opening a previously-saved session or illuminator.

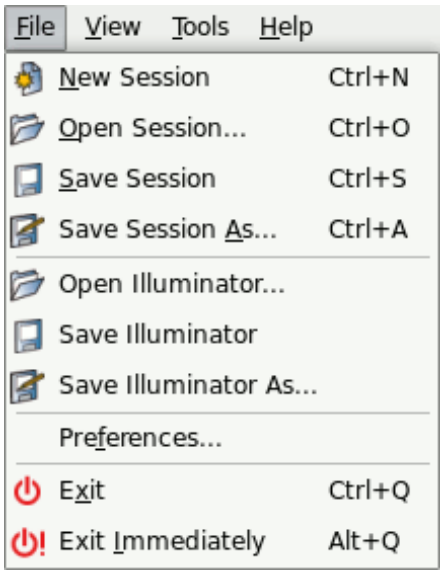


Figure 5-2. File Menu

The following paragraphs describe the options on the File menu in more detail.

New Session

Mnemonic: N
Accelerator: Ctrl+N

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 session but have not yet been saved, NightLight will ask you if you wish to save the current session before proceeding.

Open Session

Mnemonic: O
Accelerator: Ctrl+O

Launches a standard file selection dialog which allows you to specify a previously-saved session file.

If changes have been made to the current session but have not yet been saved, NightLight will ask you if you wish to save the current session before proceeding.

Save Session

Mnemonic: S
Accelerator: Ctrl+S

Saves the current session to a session configuration file quickly.

You are not prompted for the filenames where the session is to be saved. It is automatically saved to the same file it was opened from or previously saved to.

If the current session has not been saved to a file in the past, a Save Session As action will be done.

Save Session As

Mnemonic: A
Accelerator: Ctrl+A

Launches a standard file selection dialog which allows you to specify the filename where the session will be saved

Open Illuminator

Launches a standard file selection dialog which allows you to specify an illuminator's **`config.xml`** file to edit.

If changes have been made to the current illuminator but have not yet been saved, NightLight will ask you if you wish to save the current illuminator before proceeding.

The illuminator is opened in the Editor page (or window), but is not added to the session. To add an illuminator to the session, open the illuminator through the context menu on the Create, Customize, and Build branch of the Manager page (or window).

Save Illuminator

Saves the current illuminator to a **`config.xml`** file quickly (see "Illuminator Files" on page 5-66).

You are not prompted for the filename where the illuminator is to be saved. It is automatically saved to the same file it was opened from or previously saved to.

Save Illuminator As

Launches a standard file selection dialog which allows you to specify the filename where the illuminator's **`config.xml`** will be saved (see "Illuminator Files" on page 5-66).

Preferences...

Mnemonic: F

This option launches the Preferences Dialog which allows you to specify preferences for NightLight, including font selection.

Saved user preferences are applied to all NightLight invocations for the user. Preferences are saved in the user's home directory and have a broader application than session configuration files.

See "Preferences Dialog" on page 8-40 for more information.

Exit

Mnemonic: X
Accelerator: Ctrl+Q

Closes the session and exits NightLight completely.

If changes have been made to the current session or illuminator but have not yet been saved, NightLight will ask you if you wish to save the session or illuminator before exiting.

Exit Immediately

Mnemonic: I
Accelerator: Alt+Q

Closes the session and illuminator and exits NightLight without prompting to save changes that have been made. Any changes will be lost.

View

Accelerator: Alt+V

The View menu contains items for controlling the appearance of Console, Editor, and Wizard pages (or windows) of the graphical user interface. The Console page (or window) captures output from external commands that NightLight invokes. The Editor page (or window) is used to customize an illuminator. The Wizard page (or window) provides a simplified guide through the work flow.

Figure 5-3. View Menu

Console in Page

Accelerator: Alt+K

Toggles placing the Console window (the window to which output from invoked commands is logged) in a tabbed page within the main window.

Show Console

Mnemonic: C
Accelerator: Ctrl+K

Toggles showing or hiding the Console window (or page).

Clear Console

Clears the contents of the Console window (or page).

Editor in Page

Accelerator: Alt+E

Toggles placing the Editor window (the window in which an individual illuminator may be customized) in a tabbed page within the main window.

Show Editor

Mnemonic: E
Accelerator: Ctrl+E

Toggles showing or hiding the Editor window (or page).

Search Editor

Mnemonic: S
Accelerator: Ctrl+F

Toggles displaying the search bar in the Editor window (or page).

Search Editor Again

Mnemonic: A
Accelerator: Ctrl+G

Repeats the search in the search bar in the Editor window (or page).

Wizard in Page

Accelerator: Alt+W

Toggles placing the Wizard window (the window that provides a simpler guided interface through the workflow) in a tabbed page within the main window.

Wizard Console

Mnemonic: W
Accelerator: Ctrl+W

Toggles showing or hiding the Wizard window (or page).

Verbose Wizard

Mnemonic: V
Accelerator: Alt+V

Toggles whether the Wizard window (or page) includes verbose instructions guiding you through the workflow.

The figure below shows the main window if the Console and Editor are shown in tabbed pages, the search bar is displayed on the Editor page, and the Wizard is hidden:

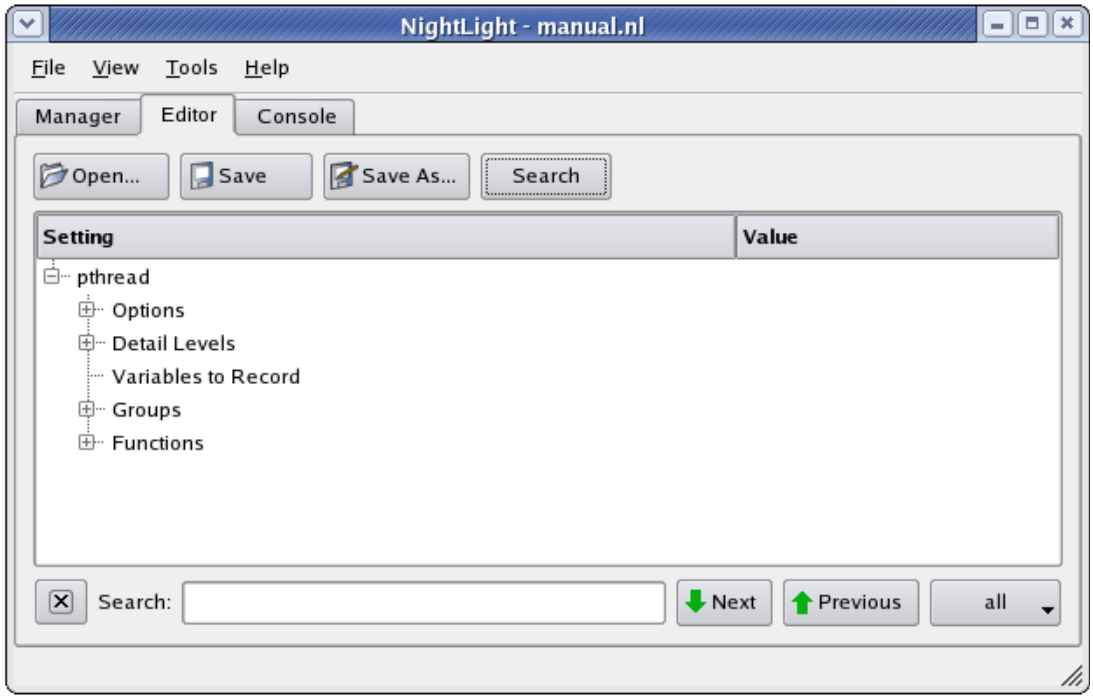


Figure 5-4. Console and Editor in Tabbed Pages with Search Bar Displayed

Tools

Mnemonic: Alt+L

Figure 5-5. Tools Menu

The following describe the options on the Tools menu:

NightProbe Monitor

Mnemonic: P

Opens the NightProbe Data Monitoring tool. 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 for data capture or to create a "control panel" for program input and output.

NightSim Scheduler

Mnemonic: S

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.

NOTE

NightSim is not available on some systems. NightSim depends on the Frequency Based Scheduler. See "Kernel Dependencies" on page B-1 for more information.

NightTune Tuner

Mnemonic: U

Opens the NightTune Tuner. NightTune is a graphical tool for analyzing the status of the system in terms of processes, interrupts, context switches, interrupt CPU affinity, processor shielding and hyper-threading control as well as network and disk

activity. NightTune can adjust the scheduling attributes of individual or groups of processes, including priority, policy, and CPU affinity.

For systems that support CPU shielding, NightTune provides a handy interface for controlling shielding, including downing sibling hyper-threaded CPUs to avoid interference.

NightView Debugger

Mnemonic: V

Opens the NightView Source-Level Debugger. NightView is a graphical source-level debugging and monitoring tool specifically designed for real-time applications and multi-threaded applications. NightView can monitor, debug, and patch multiple real-time processes running on multiple processors with minimal intrusion.

Help

Mnemonic: Alt+H

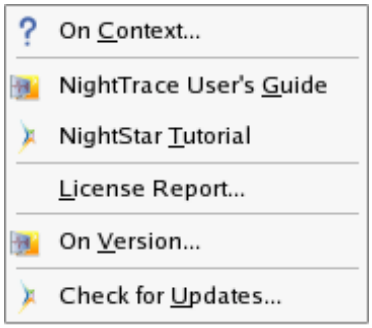


Figure 5-6. Help Menu

The following describe the options on the Help menu:

On Context

Mnemonic: C

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 Context menu item is selected). The cursor turns to a circle with a backslash when the item under the cursor has no help description associated with it.

In addition, context-sensitive help may be obtained for the currently highlighted option by pressing the F1 key. NightStar's online help system, will open with the appropriate topic displayed.

NightTrace User's Guide

Mnemonic: G

Opens the online version of the *NightTrace User's Guide* in the NightStar help viewer. The NightStar help viewer is an HTML browser which is tightly integrated with NightTrace. Help actions within NightTrace instruct the help system to show specific topics.

NightStar RT Tutorial

Mnemonic: T

Opens the online version of the *NightStar RT Tutorial* in the online help viewer.

License Report

Mnemonic: L

Opens a license dialog which indicates the current license server and the number of licenses available on the system.

On Version

Mnemonic: V

Displays a short description of the current version of NightLight.

Check for Updates...

Mnemonic: U

Launches NUU (Network Update Utility) enabling you to update your system with the latest NightStar software. This requires network access to Concurrent's Updates web site. Updates require a login and user ID issued by Concurrent. Refer to http://redhawk.ccur.com/updates for complete information.

Wizard

The wizard guides you through the basic functionality of the NightLight tool with more explicit on-screen explanations than the session manager has. It consists of a sequence of six dialogs that may be accessed randomly via the navigation panel on the left edge of each dialog, or sequentially via the Prev and Next buttons at the bottom of each dialog.

Navigation Panel

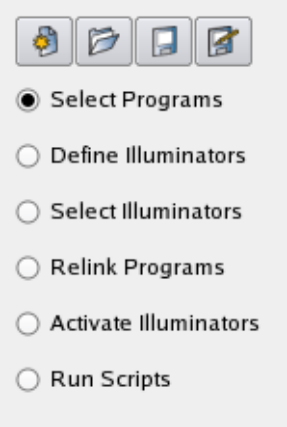


Figure 5-7. Wizard Navigation Panel

The four buttons at the top are for creating, opening, and saving sessions. These commands may also be accessed through the File menu (see “File” on page 5-6).

New Session

Creates a new session. If the current session has unsaved modifications, you will will be prompted to save it before the new session is created.

Open Session

Opens a saved session. If the current session has unsaved modifications, you will be prompted to save it before the saved session is opened.

Save Session

Saves the current session. If the session has never been saved to a file, you will be prompted for a filename to save it to.

Save Session As

Saves the current session to a new filename that you will be prompted for.

The next six buttons are radio buttons that select which of the six Wizard dialogs to display.

Select Programs

Goes directly to the Select Programs with Debug Information dialog. In this dialog, you will tell NightLight about the programs you wish to instrument with trace events. See "Select Programs with Debug Information" on page 5-18.

Define Illuminators

Goes directly to the Define an Illuminator for each Program dialog. In this dialog, you will optionally create an illuminator for the statically linked portion of each program. See "Define an Illuminator for each Program" on page 5-20.

Select Illuminators

Goes directly to the Select Predefined Illuminators for each Program dialog. In this dialog, you will select from the illuminators provided with NightTrace (`main`, `glibc`, `pthread`, and `ccur_rt`) to link with each program. See "Select Predefined Illuminators for each Program" on page 5-24.

Relink Programs

Goes directly to the Relink Illuminated Programs dialog. In this dialog, you will tell NightTrace how to relink your programs to include the user-defined and provided illuminators. See "Relink Illuminated Programs" on page 5-26.

Activate Illuminators

Goes directly to the Activate Illuminators in each Program dialog. In this dialog, you will select which illuminators linked into each program to activate, what file to record events to, and the amount of detail to record with each illuminator's events. See "Activate Illuminators in each Program" on page 5-29.

Run Scripts

Goes directly to the Run Scripts to Launch Programs and NightTrace dialog. In this dialog, you will create script(s) to run your programs and analyze the resulting events with NightTrace. See "Run Scripts to Launch Programs and NightTrace" on page 5-32.

Common Buttons

These buttons are found at the bottom of each dialog.

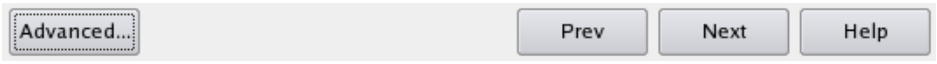


Figure 5-8. Wizard Common Buttons

Advanced...

Opens the appropriate spot in the Manager to perform more advanced operations related to the current dialog. On the Define an Illuminator for each Program dialog, this button is actually a menu of advanced operations.

Prev

Goes to the previous dialog in the workflow.

Next

Goes to the next dialog in the workflow.

Help

Gets help on the current dialog.

Select Programs with Debug Information

This dialog is used to inform NightLight about the programs that you wish to instrument with trace points.

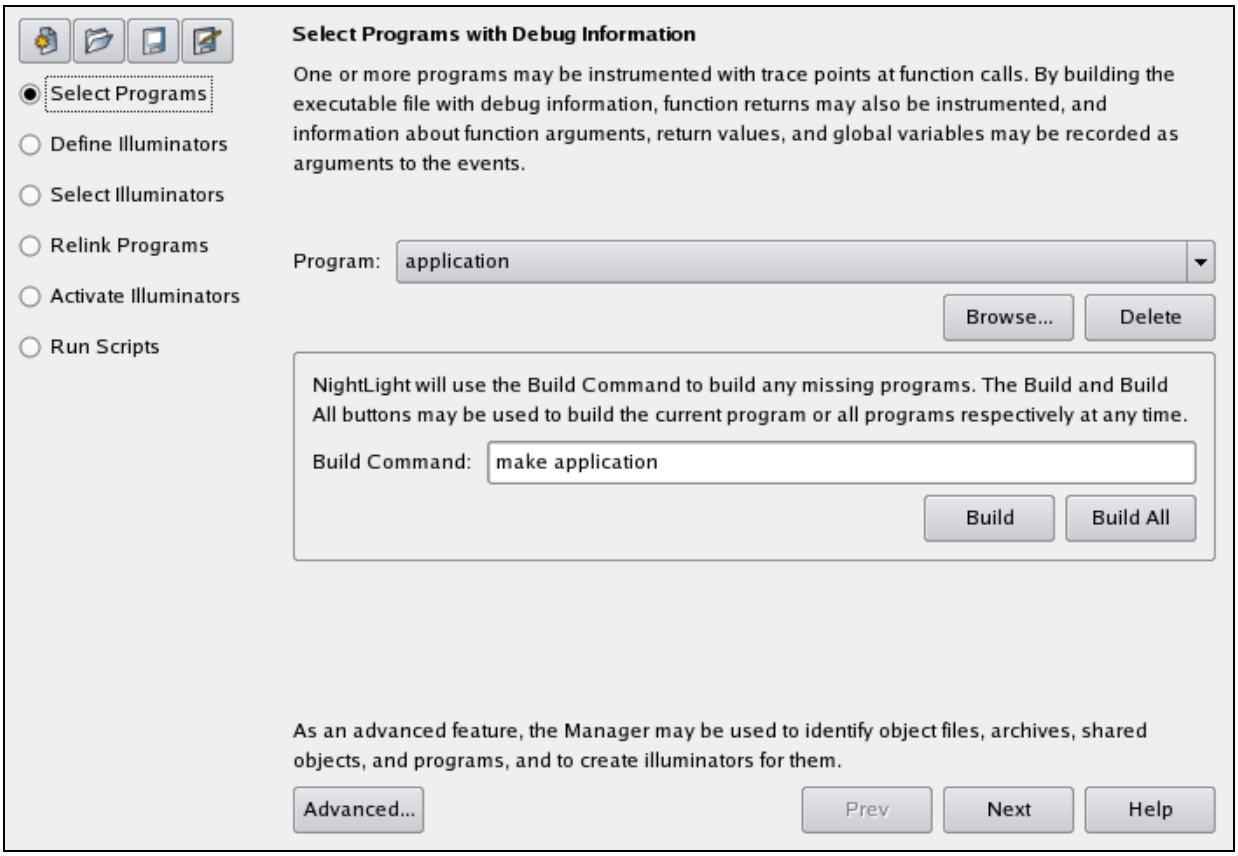


Figure 5-9. Select Programs with Debug Information Dialog

Program

Selects which program is the current program. Add or remove programs from this list with the Browse… and Delete buttons.

Browse...

Browses for another program to add to the list of programs using the standard file selection dialog. The program does not have to be built already (see Build Command, below).

Delete

Removes the current program from the list of programs. The program's executable file is not deleted.

Build Command

Specifies a command that may be used to build the current program. If the program isn't already built, NightLight will automatically invoke this command when it needs to access the program. It will not update an already built program. To update an already built program, build it at a shell prompt or use the Build or Build All buttons.

Build

Builds the current program by invoking the Build Command.

Build All

Builds all programs listed in the Program list by invoking their Build Commands.

Advanced...

Brings the session manager to the top and expands the Select Code with Debug Information branch down to the current program (see "Select Code with Debug Information" on page 5-39). There, object files, archives, and shared objects may also be selected. Illuminators may be constructed for any of these. In most situations, creating illuminators for whole programs is what you will want to do.

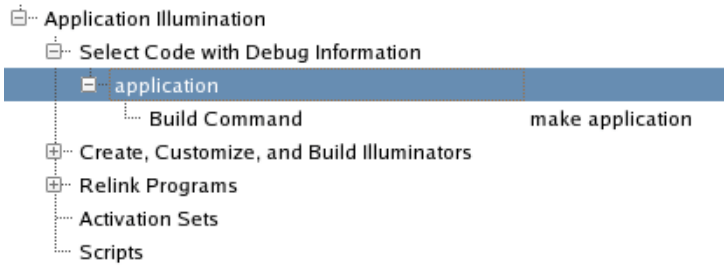


Figure 5-10. Select Programs Advanced Settings

Define an Illuminator for each Program

This dialog is used to optionally define an illuminator for the functions in the statically linked portion of each program. The default is to create the illuminator. Clear the check box to delete the illuminator. Regular expressions may be used to control which functions the illuminator will trace.

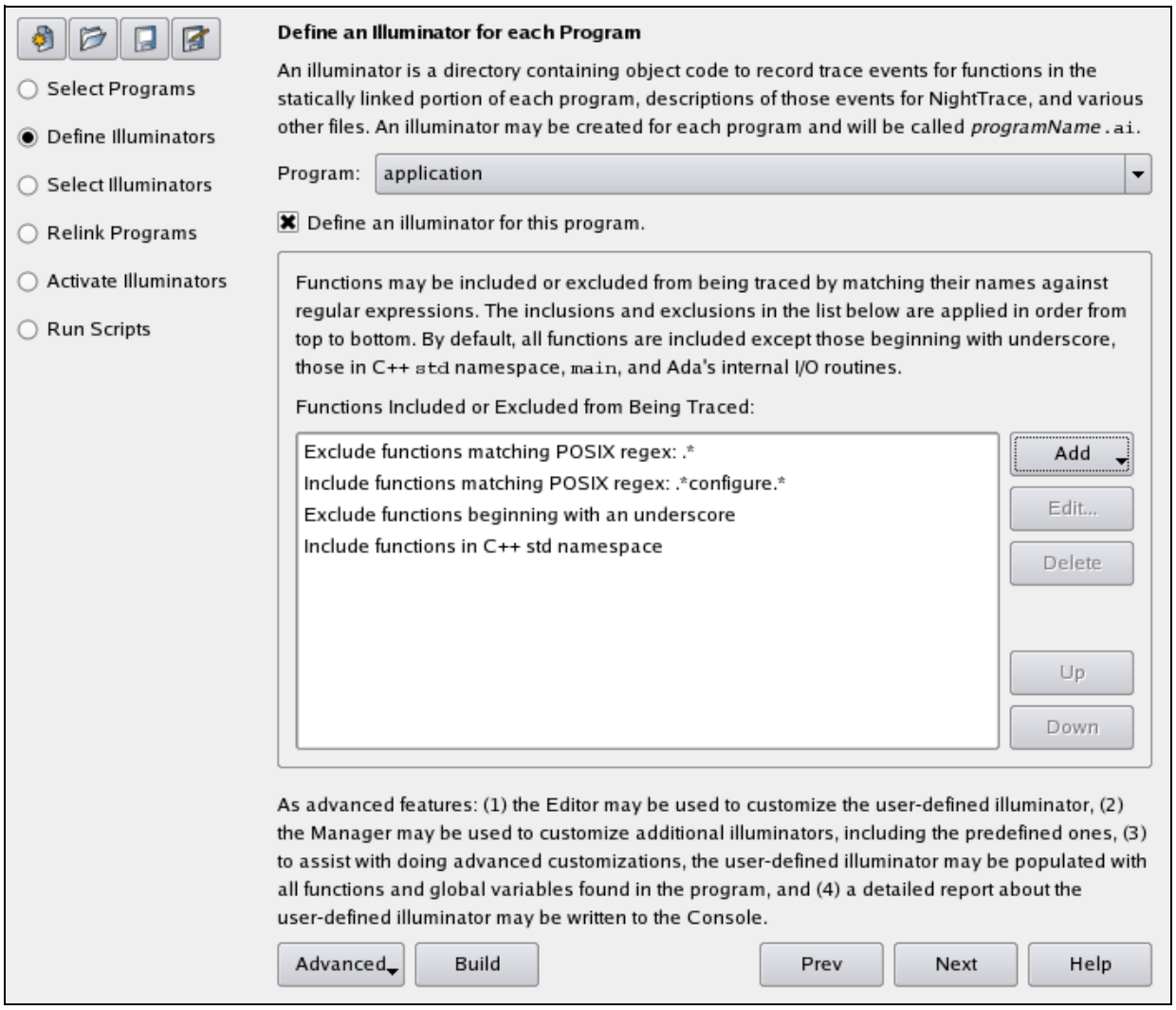


Figure 5-11. Define an Illuminator for each Program Dialog

Program

Selects the current program. To add or remove programs from this list, see "Select Programs with Debug Information" on page 5-18.

Define an illuminator for this program

Creates an illuminator to hold the code to record trace events on function entry and return for functions defined in the statically linked portion of the current program. This item will be selected by default. Clearing the checkbox will delete the illuminator. To temporarily disable the illuminator, see "Activate Illuminators in each Program" on page 5-29. The name of the illuminator will be *currentProgramName*`.ai`.

Functions Included or Excluded from Being Traced

Controls which functions are traced with a list of regular expressions that are applied in sequence from top to bottom. To restrict instrumentation to a small list of functions, first exclude all functions matching the POSIX regular expression "`.*`", then include those functions you wish to trace. See "Add" on page 5-21 for documentation on the various regular expressions available. By default, all functions are included except those beginning with underscore, those in the C++ `std` namespace, `main`, and Ada's internal routines (see "Regular Expressions" on page 5-81).

Add

Adds a regular expression that will include or exclude functions from being traced. Select the expression from the menu of choices that pop up when this button is clicked.

Include functions beginning with an underscore
Exclude functions beginning with an underscore

Includes or excludes functions whose names start whith an underscore character. All aliases of a function and the fully qualified C++ name (if applicable) must begin with an underscore in order to match these regular expressions (in contrast to POSIX regular expressions). A fully qualified C++ name matches if the function name or the name of any containing classes start with an underscore.

The rationale for this is that functions and class names that begin with underscores are typically vendor implementation routines that are of less interest. But it is also common practice to create a strongly defined function that starts with an underscore, then weakly define aliases to that function that do not. These functions, like many in Glibc, are likely to be interesting, and so aren't matched by these expressions.

The default is to exclude functions beginning with an underscore.

Include functions in the C++ std namespace
Exclude functions in the C++ std namespace

Includes or excludes C++ functions in the `std` namespace.

The default is to exclude C++ functions in the `std` namespace. Such functions are often inlined and so tracing them usually doesn't provide a lot of useful information.

Include functions matching POSIX regex
Exclude functions matching POSIX regex

Includes or excludes functions whose names match a POSIX regular expression (see **`regex(7)`**). A function name matches the regular expression if any alias or fully qualifed C++ name (if applicable) matches it. The regular expression must match the whole name (an implicit `^` and `$` are placed before and after the regular expression respectively).

By default `main` and Ada's internal I/O routines are excluded.

You will be prompted for a POSIX regular expression to type in when this menu item is selected.

Edit...

Edits the POSIX regular expression of the currently selected regular expression.

Delete

Deletes the currently selected regular expression.

Up

Moves the currently selected regular expression up one place in the list.

Down

Moves the currently selected regular expression down one place in the list.

Advanced

Provides a menu of advanced actions to choose from.

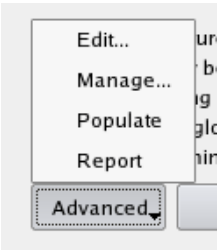


Figure 5-12. Define Illuminators Advanced Menu

Edit...

Opens the illuminator for the current program in the Editor window (or page) to perform advanced customization. See "Customizing an Illuminator with the Editor" on page 5-76

Manage...

Brings the session manager to the top and expands the Create, Customize, and Build Illuminators branch. Additional custom illuminators may be created and customized here. The provided illuminators (`main`, `glibc`, `pthread`, and `ccur_rt`) may alo be customized (requires that the debug-info packages for Glibc be installed). See “Create, Customize, and Build Illuminators” on page 5-43.

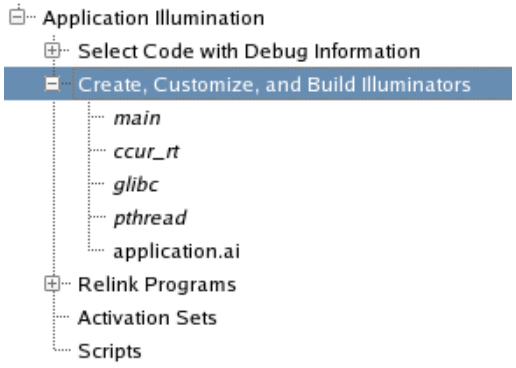


Figure 5-13. Define Illuminators Advanced Settings

Populate

Populates the illuminator for the current program with the functions and variables found in the program. It is not necessary to populate an illumintor to customize, build, or use it. Populating an illuminator can be convenient for making many customizations to it. See “Populate” on page 5-45 and “nlight --populate” on page 5-71.

Report

Creates a report about the functions being traced by the illuminator for the current program. The report is written to the Console window (or page). See “nlight --report” on page 5-72.

Build

Builds the illuminator. If an illuminator’s `config.xml` file or the program or object files it illuminates have changed, NightLight will update the illuminator anytime it needs to access its files. Therefore, it is normally not necessary for you to use this button. However, initiating the build manually is useful to verify that customizations done through the Editor window (or page) will build successfully.

Select Predefined Illuminators for each Program

This dialog is used to select predefined illuminators to link into the illuminated program (in addition to the user-defined illuminator created in the previous dialog). See "Predefined Illuminators" on page 5-64.

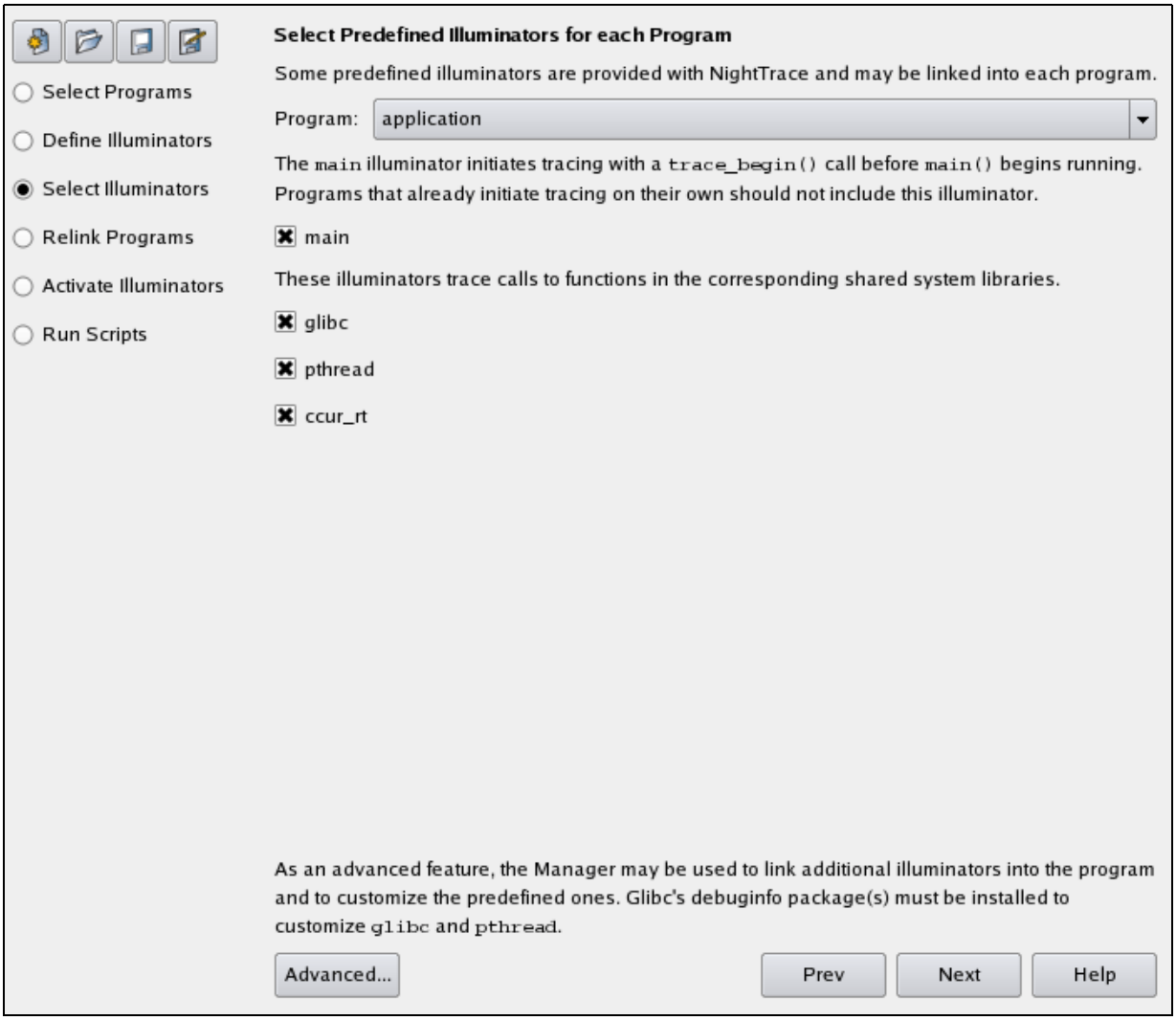


Figure 5-14. Select Predefined Illuminators for each Program Dialog

Program

Selects the current program. To add or remove programs from this list, see "Select Programs with Debug Information" on page 5-18.

main

Links the `main` illuminator into the current program, which does not record any events, but calls `trace_begin()` before `main()` is called. This is necessary if the traced program does not do its own `trace_begin()` call. Do not use the `main` illuminator in programs that already call `trace_begin()` on their own. See “main” on page 5-64.

glibc

Links the `glibc` illuminator into the current program, which illuminates calls to the system C library. See “glibc” on page 5-64.

pthread

Links the `pthread` illuminator into the current program, which illuminates calls to the system POSIX threads library. See “pthread” on page 5-65.

ccur_rt

Links the `ccur_rt` illuminator into the current program (if available), which illuminates calls to the Concurrent real-time library. See “ccur_rt” on page 5-65.

Advanced...

Brings the session manager to the top and expands the Create, Customize, and Build Illuminators branch and the Relink Programs branch down to the Illuminators list of the current program. Additional illuminators may be created or linked with programs. See “Create, Customize, and Build Illuminators” on page 5-43 and “Relink Programs” on page 5-47.

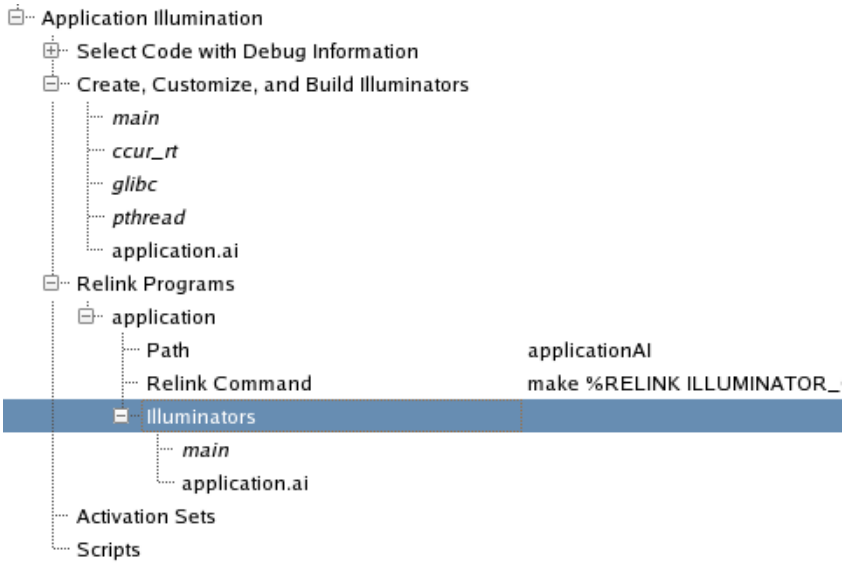


Figure 5-15. Select Illuminators Advanced Settings

Relink Illuminated Programs

This dialog is used to link a copy of each program to include the code from the illuminator(s) to record the events.

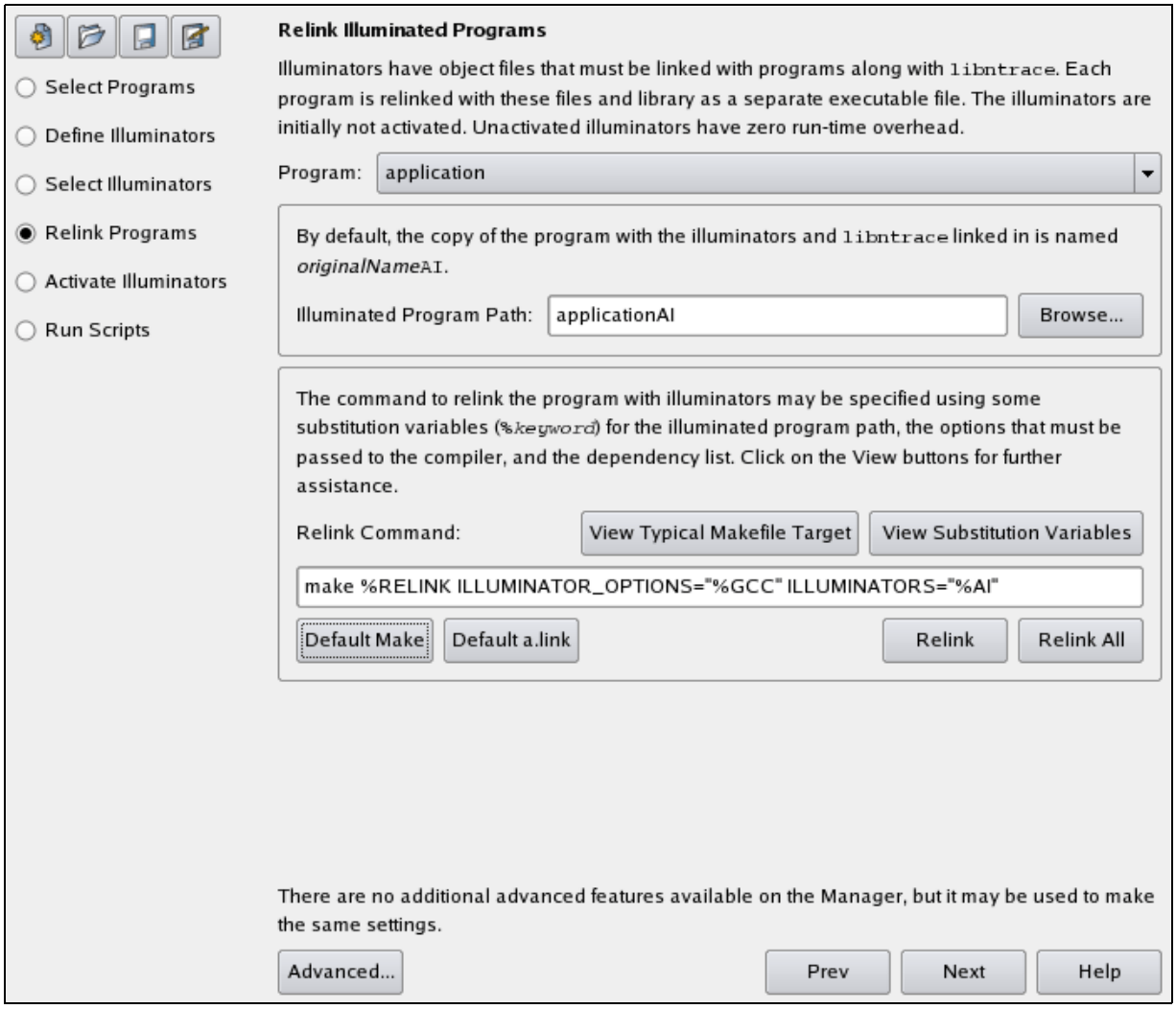


Figure 5-16. Relink Illuminated Programs Dialog

Program

Selects the current program. To add or remove programs from this list, see "Select Programs with Debug Information" on page 5-18.

Illuminated Program Path

Specifies the path name of the illuminated copy of the program. The original program is relinked with the illuminators specified for it in the previous two dialogs and

is given a distinct name. By default, it is called *originalProgramPath***AI**. See "Path" on page 5-48.

Browse

Browses the file system using the standard file selection dialog for the Illuminated Program Path.

Relink Command

Specifies the external command to relink the program. By default, it is a **make** command using the Illuminated Program Path as the target name. There are a number of substitution variables that may be specified in the command. These begin with the "**%**" character and are replaced by NightLight when the command is invoked. See "Relink Command" on page 5-48

View Typical Makefile Target

Displays a typical **Makefile** target assuming the default **make** command. You will need to modify your **Makefile** to include the Illuminated Program Path as a target.

View Substitution Variables

Displays a list and brief description of the available substitution variables for the Relink Command.

%RELINK

Substituted with the Relink Path value. It is handy to use as a **make** target or as the operand of a **-o** option in **a.link**, **gcc**, or other compiler.

%AI

Substituted with the full paths of the illuminators to be linked in for use as a **make** file target's dependency list. The default **make** command passes **%AI** to **make** using the variable ILLUMINATORS.

%GCC, %G77, %CF77, %ADA

Substituted with the options, files, and libraries that are needed to link with the illuminators using the **gcc**, **g77**, **cf77**, or **a.link** commands (respectively). This includes the NightTrace library. The default **make** command passes **%GCC** to **make** using the variable ILLUMINATOR_OPTIONS.

Default Make

Sets the Relink Command to the default **make** command.

Default a.link

Sets the Relink Command to the default **`a.link`** command (for Ada programs).

Relink

Relinks the current program by invoking the Relink Command. NightLight will automatically relink your program (and apply the default activation set to it) whenever it is out-of-date and the relinked program is needed. The Relink button is useful to test changes to the Relink Command right away.

Relink All

Relinks all programs listed in the Program list by invoking their Relink Commands.

Advanced...

Brings the session manager to the top and expands the Relink Programs branch down to the current program. There are no additional features here to access. See "Relink Programs" on page 5-47.

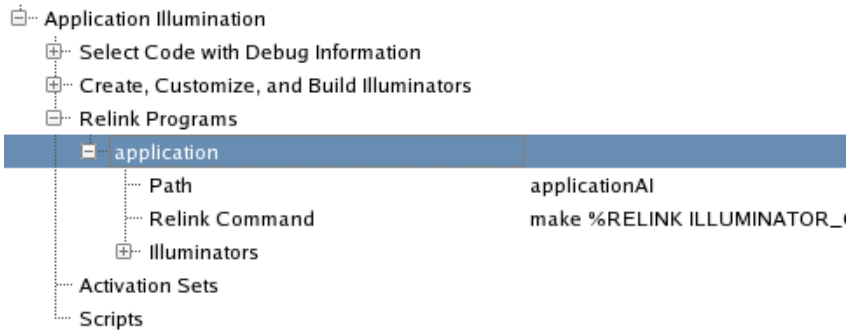


Figure 5-17. Relink Programs Advanced Settings

Activate Illuminators in each Program

This dialog is used to activate illuminators so that they record events. Illuminators are "inert", having no run-time overhead and recording no events, when first linked into a program. They must first be activated. Check the box next to each illuminator you want activated. Only those illuminators that are actually linked into the program will appear in this dialog.

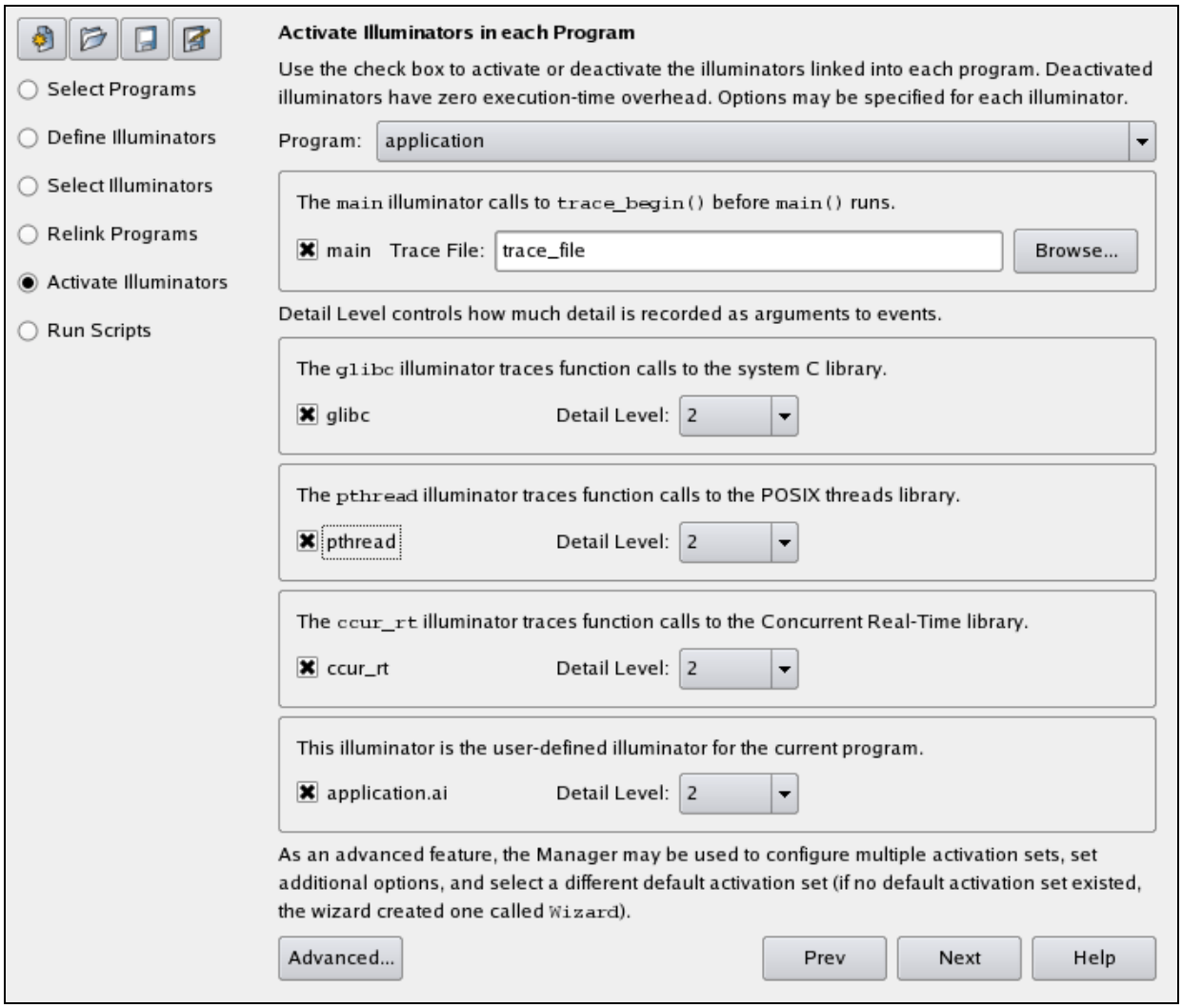


Figure 5-18. Activate Illuminators in each Program Dialog

main
glibc
pthread

ccur_rt
currentProgram.**ai**

Enables (if checked) or disables (if not checked) an illuminator. Only predefined or the user-defined illuminators that are actually linked into the illuminated program are listed. Additional custom illuminators added as an advanced feature in the session manager can only be enabled or disabled from the session manager. A notice will appear in the dialog if such illuminators are linked in.

Figure 5-19. Notice That Additional Illuminators Are Linked In

Program

Selects the current program. To add or remove programs from this list, see "Select Programs with Debug Information" on page 5-18.

Trace File

Specifies the file that events will be recorded in. This is a parameter to the `trace_begin()` call that the `main` illuminator does.

Browse...

Browses for the Trace File using the standard file selection dialog.

Detail Level

Specifies the level of detail that will be recorded as arguments to the events recorded by each illuminator. See "Detail Levels" on page 5-64 and "Detail Levels" on page 5-85.

Advanced...

Brings the session manager to the top and expands the Activation Sets branch down through the default activation set. A different activation set may be designated as the default. The Wizard always manipulates the default activation set. If no default activation set has been designated, the Wizard will create one called Wizard. See "Activation Sets" on page 5-52.

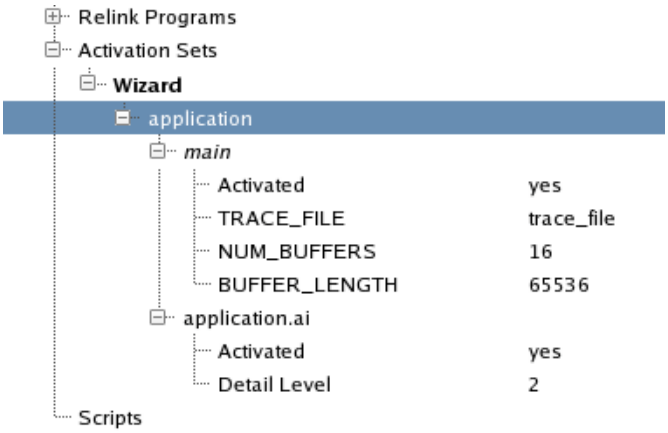


Figure 5-20. Activation Sets Advanced Settings

Run Scripts to Launch Programs and NightTrace

This dialog is used to run scripts for collecting and analyzing trace data. NightTrace may collect data from programs in two ways. In File mode, your programs communicate with daemons to log events to a file on disk, then NightTrace is used to analyze those events. In Stream mode, your programs stream events directly to a running NightTrace. Simple scripts are automatically generated, and may then be customized, to run NightTrace and your programs in these two modes. See "Scripts" on page 5-58.

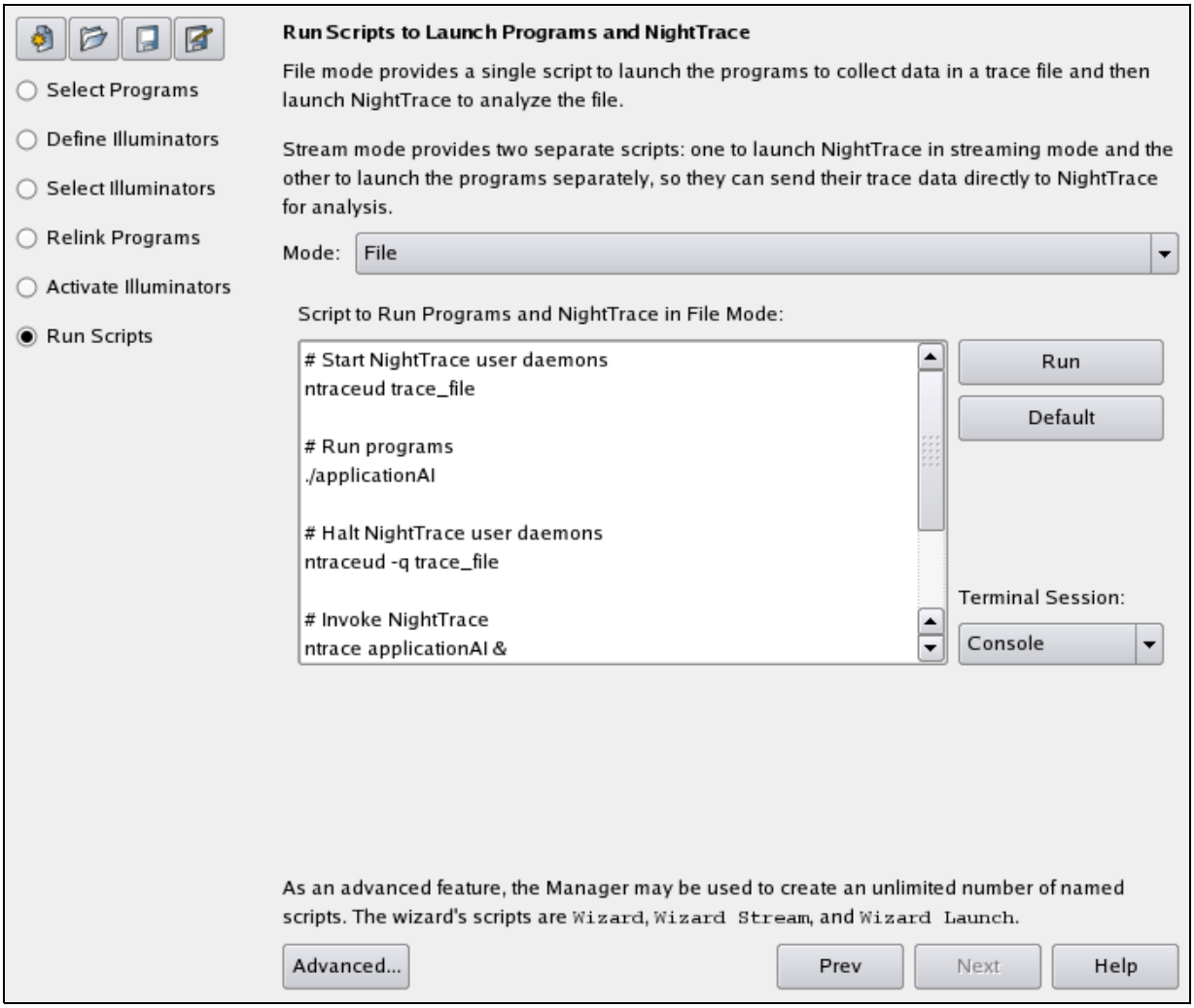


Figure 5-21. Run Scripts to Launch Programs and NightTrace Dialog in File Mode

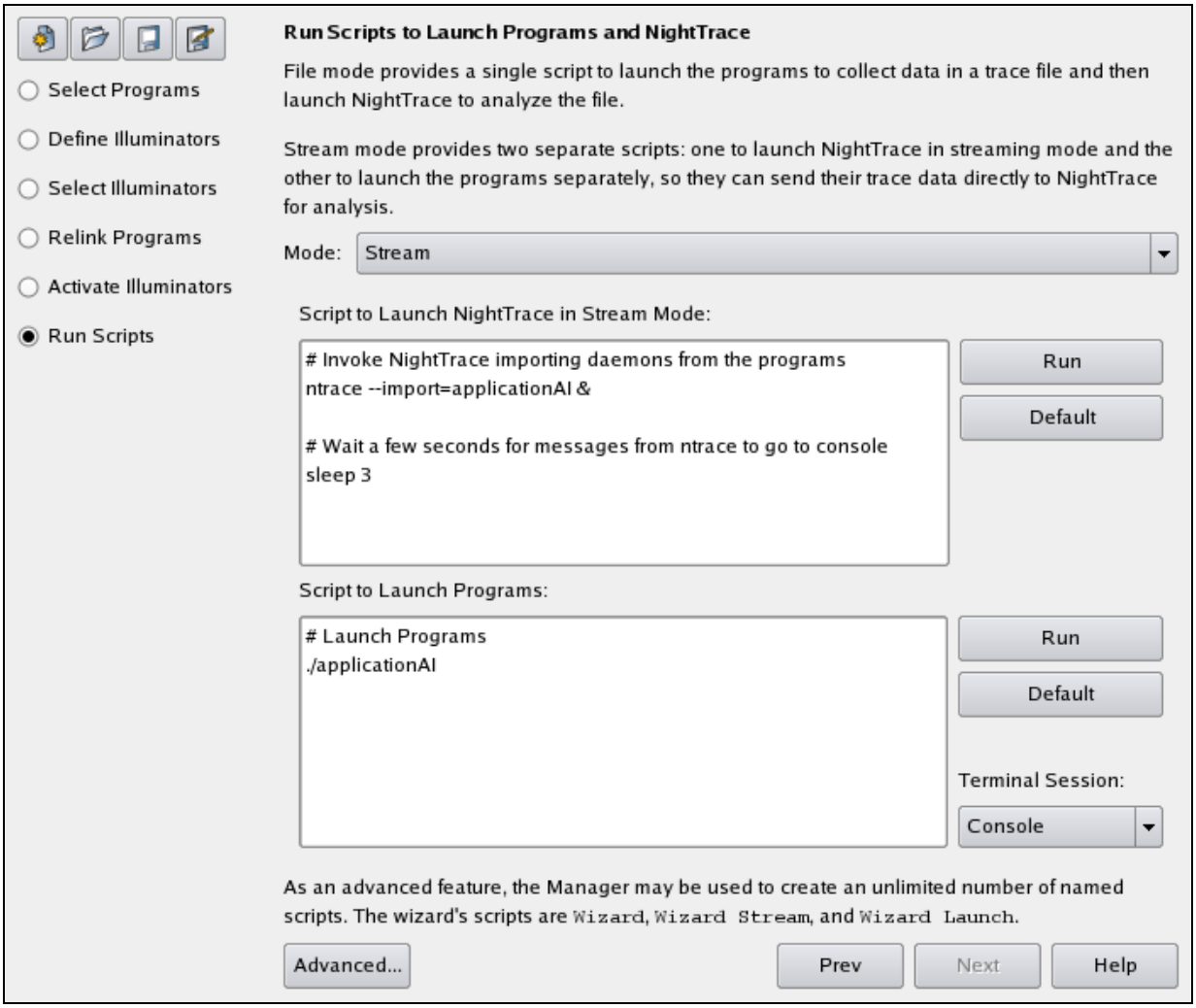


Figure 5-22. Run Scripts to Launch Programs and NightTrace Dialog in Stream Mode

Mode

Selects between File mode and Stream mode. The dialog reconfigures itself to show the scripts appropriate to each mode.

In File mode, a single script (called Wizard in the session manager) is generated that will start daemons to record events in files, run your programs, stop the daemons, and run NightTrace on the trace files.

In Stream mode, two scripts (called Wizard Stream and Wizard Launch in the session manager) are generated. The first will run NightTrace in stream mode, and the second will run your programs.

Script to Run Programs and NightTrace in File Mode

Launches (for File mode) user daemons for all your programs, runs your programs in sequence, halts the daemons, and runs NightTrace on the resulting trace files. NightLight only knows the daemons to launch for programs that use the `main` illuminator to do the `trace_begin()` call. For other programs, you will need to modify the script to launch them yourself.

If you add or remove programs, change the path to any of the relinked programs, or change the file events are recorded in, you can recreate the script by clicking on the Default button. Any edits you've done will be lost when you do this. In the session manager, this is the Wizard script.

Script to Launch NightTrace in Stream Mode

Launches (for Stream mode) NightTrace in stream mode. You may then launch, start, and stop the daemons from within NightTrace. NightTrace only knows the daemons needed for programs linked with the `main` illuminator. For other programs, it will prompt for the daemon name. Events will stream directly into NightTrace when your programs are launched with the following script.

If you add or remove programs, change the path to any of the relinked programs, or change the file events are recorded in, you can recreate the script by clicking on the Default button. Any edits you've done will be lost when you do this. In the session manager, this is the Wizard Stream script.

Script to Launch Programs

Launches (for Stream mode) your programs in sequence. If NightTrace has been launched and used to start the daemons, events from these programs will stream directly into NightTrace.

If you add or remove programs or change the path to any of the relinked programs, you can recreate the script by clicking on the Default button. Any edits you've done will be lost when you do this. In the session manager, this is the Wizard Launch script.

Run

Runs the adjacent script using **`/bin/sh`**. The output from the script is written to the Console page (or window) by default. The scripts that launch your programs may optionally be run in other terminal sessions, such as an **`xterm`** by using the Terminal Session setting next to the script.

Default

Resets the adjacent script to a default value that is based on the current list of programs defined and options set for them. Any edits you've done will be lost when you do this.

Terminal Session

Selects from a menu of terminal sessions in which the adjacent script may be run.

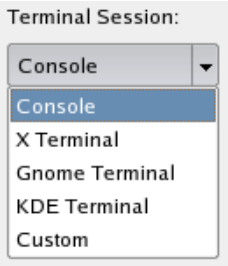


Figure 5-23. Terminal Session Menu

Console

Captures all output from the adjacent script in the Console page (or window). This is inconvenient if the program needs to get input from the user.

X Terminal
Gnome Terminal
KDE Terminal

Selects various kinds of virtual terminals in which to run the adjacent script. These are convenient if the program needs to get input from the user or must run in a terminal emulator.

Custom

Selects running the adjacent script using the Custom Terminal Session Command (which may only be modified through the Advanced… button). It defaults to being an X Terminal.

Advanced...

Brings the session manager to the top and expands the Scripts branch(s) for the current mode's script(s). See "Scripts" on page 5-58.

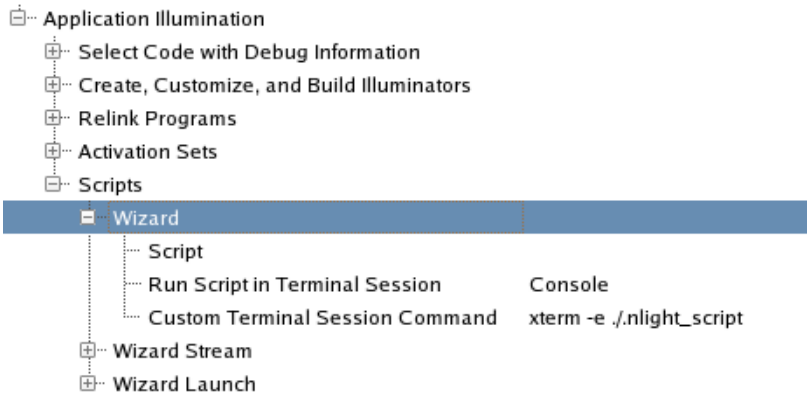


Figure 5-24. Run Scripts Advanced Settings

Session Manager

The session manager guides you through the five-step work flow. Each branch of the tree structure represents one step. Hovering over each step will bring up a tool tip describing the step. Use context menus on each item of the tree to configure and execute each step. Click on the ⊞ symbol to expand branches of the tree. Values in the Value column may be edited in place by clicking on them. Settings with Edit items in their context menus can usually be edited by double clicking on them.

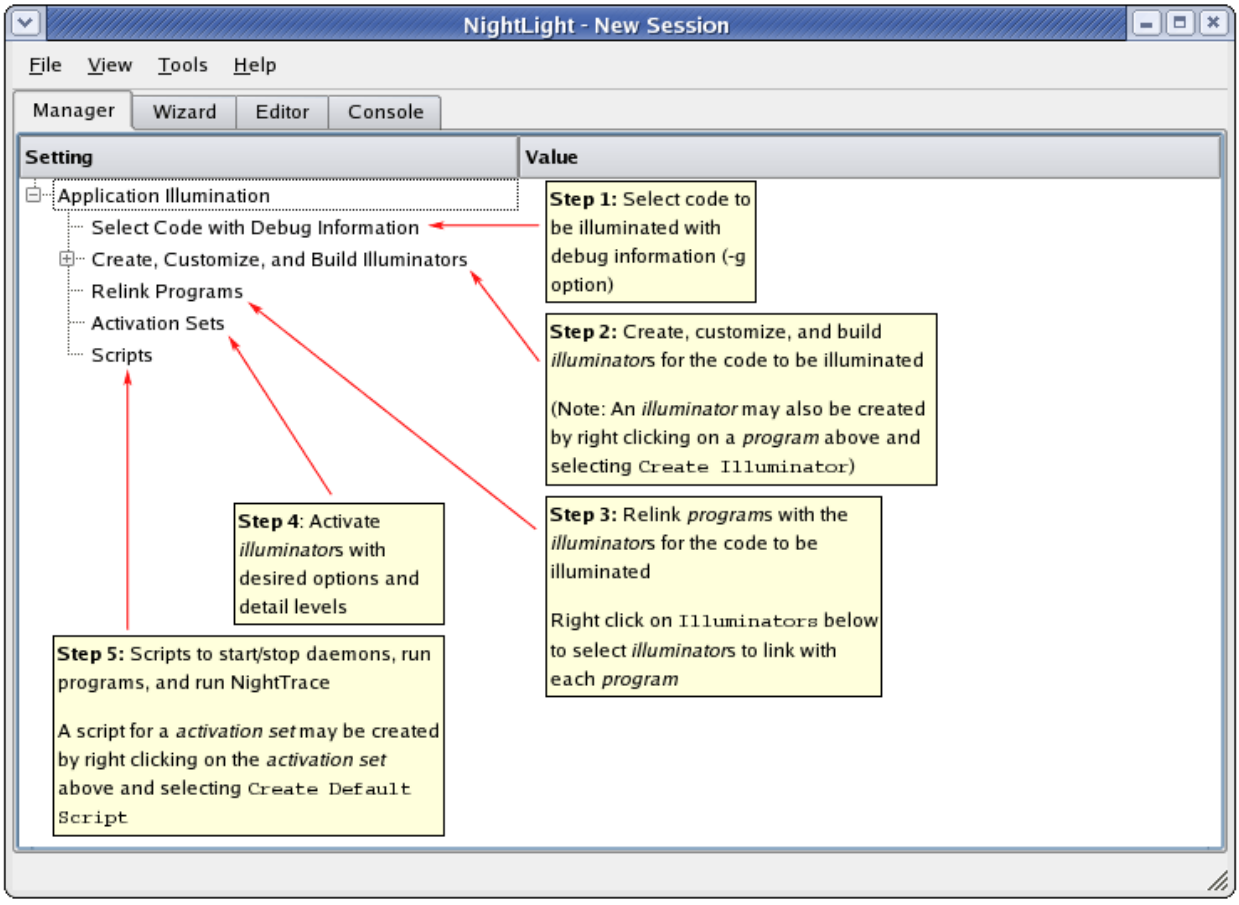


Figure 5-25. Tool Tips in the Session Manager

The root item in the Application Illumination tree displays a context menu when you right click on it.

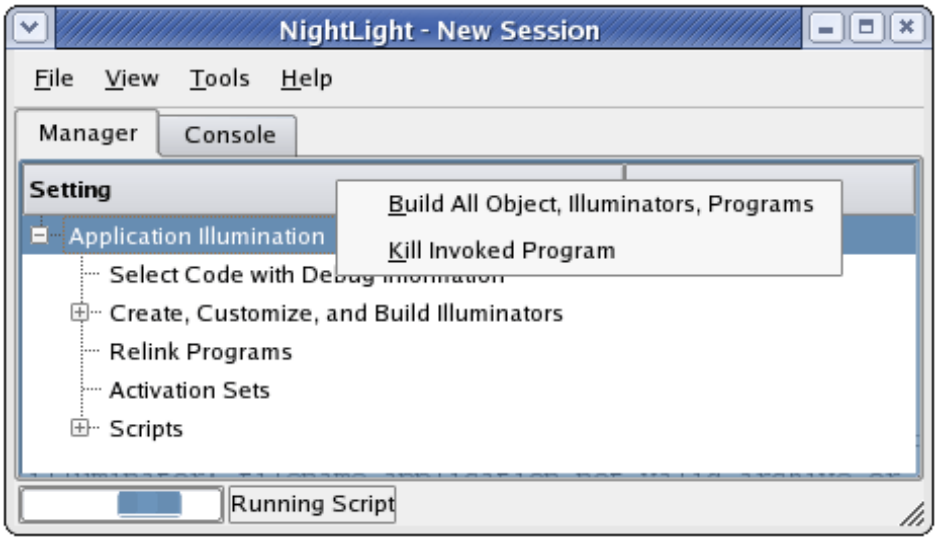


Figure 5-26. Application Illumination Context Menu

Build All Objects, Illuminators, Programs

Update steps 1-3 of the work flow. Also, if there is a default activation set (see "Make Default Activation Set" on page 5-57), that is applied to each relinked program.

Kill Invoked Program

Kill any program that NightLight has invoked to perform a task. This might be necessary if a user program or script (as in the illustration above) has entered an infinite loop. Note the busy indicator in the above illustration at the bottom of the window.

Select Code with Debug Information

The first step in the NightLight workflow is to select the code to have function entry and return events generated (that is, to be *illuminated*). NightLight can debug information to generate the illuminators and descriptions of the events that will be traced. These events can record values of parameters, global variables, return values, etc. The debug information is needed to know the names, types, and locations of these values.

Context Menus

Right click on Select Code with Debug Information to inform NightLight about objects and to build them.

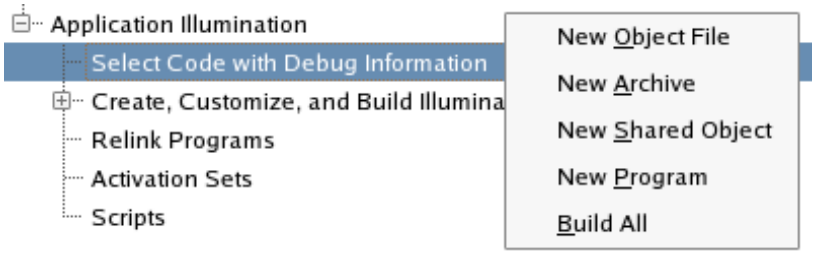


Figure 5-27. Build Code with Debug Information Context Menu

New Object File
New Archive
New Shared Object
New Program

Tells NightLight about the various kinds of objects that you will be creating illuminators for.

Build All

Builds all the objects (each object must have a Build Command configured for it for this to work; otherwise, you should build the objects outside of NightLight control).

Associated with each object is a build command. The default command that is filled in is a simple **`make`** command. Shared objects may optionally have a separate object file containing the debug unformation, called a *debug info* file. Programs also automatically get an entry in the Relink Programs section (step 3 of the workflow, see “Relink Programs” on page 5-47).

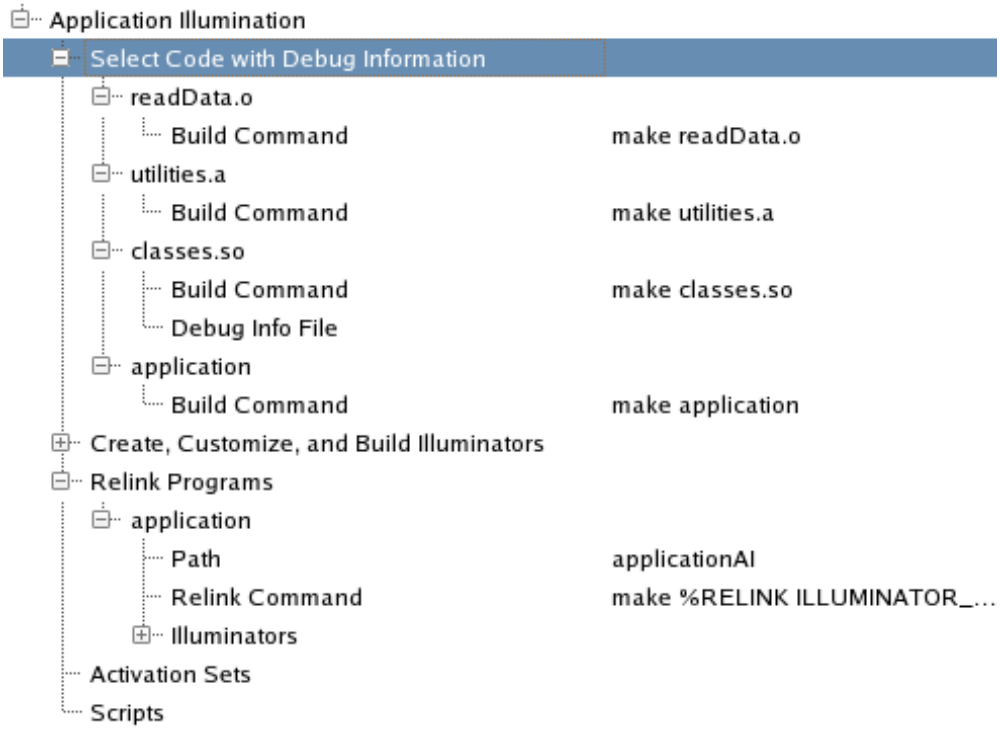


Figure 5-28. Various Objects Added to the Session Manager

The context menu for each object may be used to create an illuminator for the functions in that object, build that object, rearrange that object, or remove that object from the session manager (removing does not delete the file).

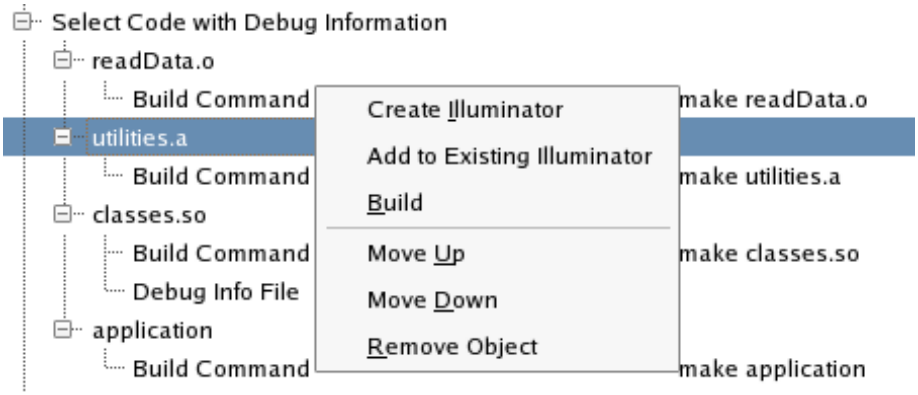


Figure 5-29. Context Menu on an Object

Create Illuminator

Creates an illuminator for the functions in this object.

Add to Existing Illuminator

Causes an existing illumintator (selected in a dialog from those listed in the Create, Customize, and Build Illuminators section) to also illuminate the functions in this object.

Build

Builds this object using the Build Command.

Move Up
Move Down

Changes the order of objects in this branch.

Remove Object

Removes all references to this object in NightLight. It does not remove the actual file, nor does it remove references to the object in any illuminators.

To edit the build command, you may double click on the command itself and edit it in place, double click on Build Command and edit it in a dialog, or use the Edit Build Command item in Build Command's context menu.

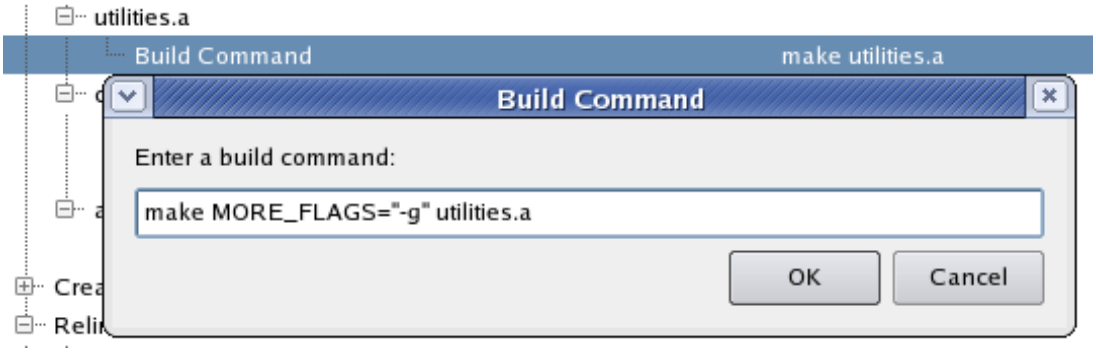


Figure 5-30. Build Command Dialog

The context menu on Debug Info File may be used to bring up the file selector dialog to specify a debug info file or to clear the debug info file setting.

Figure 5-31. Debug Info File Context Menu

Building Object

When NightLight performs an action that requires accessing an object, such as building an illuminator for it, and that object has not been built yet, NightLight will invoke its Build Command automatically.

However, since NightLight knows nothing about the build dependencies of an object, it will not automatically rebuild an object if it is stale. You must do this manually. This may be done several ways:

- Build the object outside of NightLight;
- Right click on the object in the Select Code with Debug Information section and select Build from the context menu that pops up;
- Right click on Select Code with Debug Information and select Build All from the context menu that pops up; or,
- Right click on Application Illumination (the root item in the tree) and select Build All Objects, Illuminators, and Programs from the context menu that pops up.

Create, Customize, and Build Illuminators

The Create, Customize, and Build Illuminators section is pre-populated with the four predefined illuminators:

- main, which sets up a `trace_begin()` call, and
- ccur_rt, glibc and pthread, which trace functions in the corresponding system libraries.

The predefined illuminators are displayed in an italic font unless they have been customized. A customized predefined illuminator is copied to the current working directory and is displayed using a plain roman font.

When building illuminators, NightLight will attempt to assign each illuminator whose upper Event IDs range is the maximum value (32767) a unique block of IDs starting at 12000 and working its way up in increments of 100 IDs.

Context Menu

The context menu on the root item of this section may be used to create, open, populate, or build illuminators.

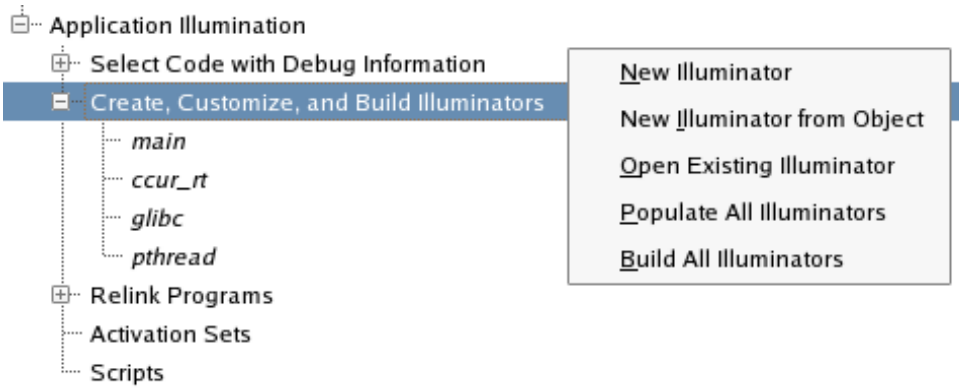


Figure 5-32. Create, Customize, and Build Illuminators Context Menu

New Illuminator

Creates a new illuminator. A file dialog will prompt you for the illuminator's name. A directory of that name will be created with a **`config.xml`** file in it.

This illuminator must be customized and built before it can be used because the created **`config.xml`** file will not specify any object file to search for functions to illuminate.

New Illuminator from Object

Creates a new illuminator. A dialog will prompt for an object containing the functions to be illuminated. Then a file dialog will prompt for the illuminator's name. A directory of that name will be created with a **`config.xml`** file in it.

An easier way to achieve the same result is to right click on the object in the Select Code with Debug Information section and select Create Illuminator from the context menu that pops up.

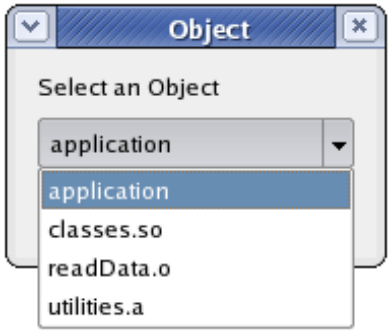


Figure 5-33. New Illuminator from Object Dialog

Open Existing Illuminator

Opens an illuminator created in another session, with a command line option, or by manually creating a directory containing a **`config.xml`** file.

Populate All Illuminators

Populates all non-predefined illuminators with functions and variables found in their objects. It is not necessary to populate an illuminator to customize, build, or use it. Populating an illuminator can be convenient for making customizations to it.

Build All Illuminators

Builds all non-predefined illuminators. An illuminator must be built before using it. It is not necessary to explicitly build illuminators. They will be updated if necessary when they are used. This context menu item is useful mainly for convenience when debugging customizations (it is possible for a customization to result in an error at build time).

Context Menu on Individual Illuminators

The context menu on an individual illuminator may be used to populate, edit (that is, customize), build, rearrange, or delete the illuminator.

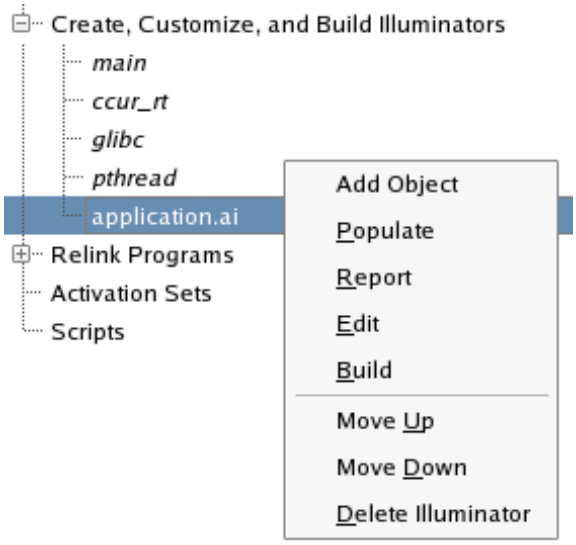


Figure 5-34. Context Menu on an Individual Illuminator

Add Object

Adds the functions from an additional object to the list of functions this illuminator will illuminate.

Populate

Populates a non-predefined illuminator with functions and variables found in its objects. It is not necessary to populate an illuminator to customize, build, or use it. Populating an illuminator can be convenient for making customizations to it.

Report

Creates a report about all variables and functions found, and what groups the functions are in, on the Console window (or page).

Edit

Customizes an illuminator by editing its **`config.xml`** file in the Editor window (or page). If the illuminator is a predefined illuminator, a copy of it is made in the current working directory and it is this copy that is customized. The italic font used for predefined illuminators is changed to a plain roman font.

Build

Builds a non-predefined illuminator. An illuminator must be built before using it. It is not necessary to explicitly build illuminators. They will be updated if necessary when they are used. This context menu item is useful mainly for convenience when debugging customizations (it is possible for a customization to result in an error at build time).

Move Up,
Move Down

Rearranges the order of the illuminators in this section.

Delete Illuminator

For customized predefined illuminators

Deletes the customized illuminator from the current working directory. All references to this illuminator revert to the pre-defined illuminator and the font used to display the name of this illuminator is changed back to italic.

For predefined illuminators

Deletes all references to the pre-defined illuminator. The predefined illumintor will remain in the list.

For non-predefined illuminators

Deletes the illuminator from the disk and removes all references to it from the session manager, including from this section.

Relink Programs

The third step in the workflow is to relink the programs, this time including the illuminators that have been built. The Relink Programs section is populated automatically with the programs that are in the Select Code with Debug Information section (see “Select Code with Debug Information” on page 5-39).

Context Menus

The context menus for Relink Programs and for individual programs under that are fairly simple. You can relink all the programs or relink individual ones. Programs are relinked if they are stale when they are needed, so it should rarely be necessary to explicitly relink them unless you are using the relinked programs outside of the control of the NightLight GUI. All programs may also be relinked by choosing the Build All Object, Illuminators, Programs item in the context menu on the root Application Illumination item. If there is a default activation set, it will be applied when the program is relinked (see “Activation Sets” on page 5-52).

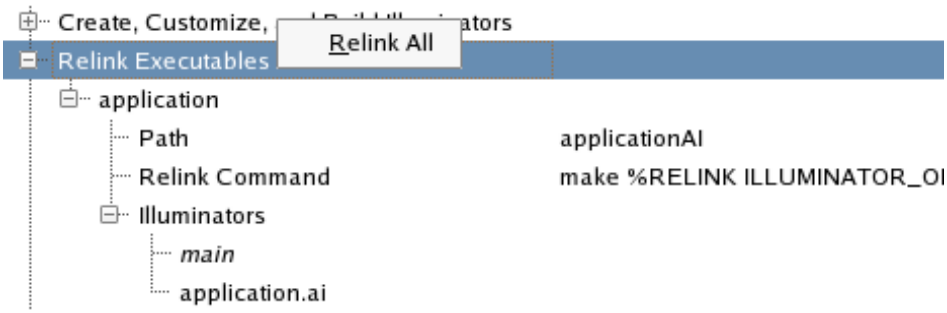


Figure 5-35. Relink Programs Context Menu

Relink All

Relinks all the programs and applies the default activation set (if there is one).

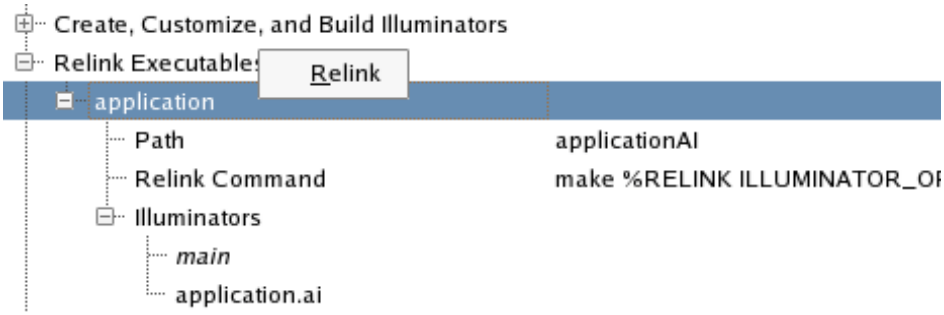


Figure 5-36. Individual Relinked Program Context Menu

Relink

Relinks this single program and applies the default activation set.

Path

The Path setting under individual programs is the file name of the relinked copy of the program. By default, it is the path of the original program (without illuminators linked in) with the capital letters "**AI**" appended.

The Path setting my be edited by double clicking on it or by right clicking on it and choosing the Edit Relink Path context menu item. A file selection dialog will display allowing you to select a new file path.

Relink Command

The Relink Command is the external command that will be used to relink the program. By default it is a **make** command. There are a number of *substitution variables* that may be specified in the command. These begin with the "**%**" character and are replaced by NightLight when the command is invoked (see below for details).

There are a number of ways the Relink Command can be edited:

- Double click on the value of the Relink Command: this allows the command to be edited in place;

Figure 5-37. Editing Relink Command In Place

- Double click on the Relink Command label: this pops up a line editing dialog;

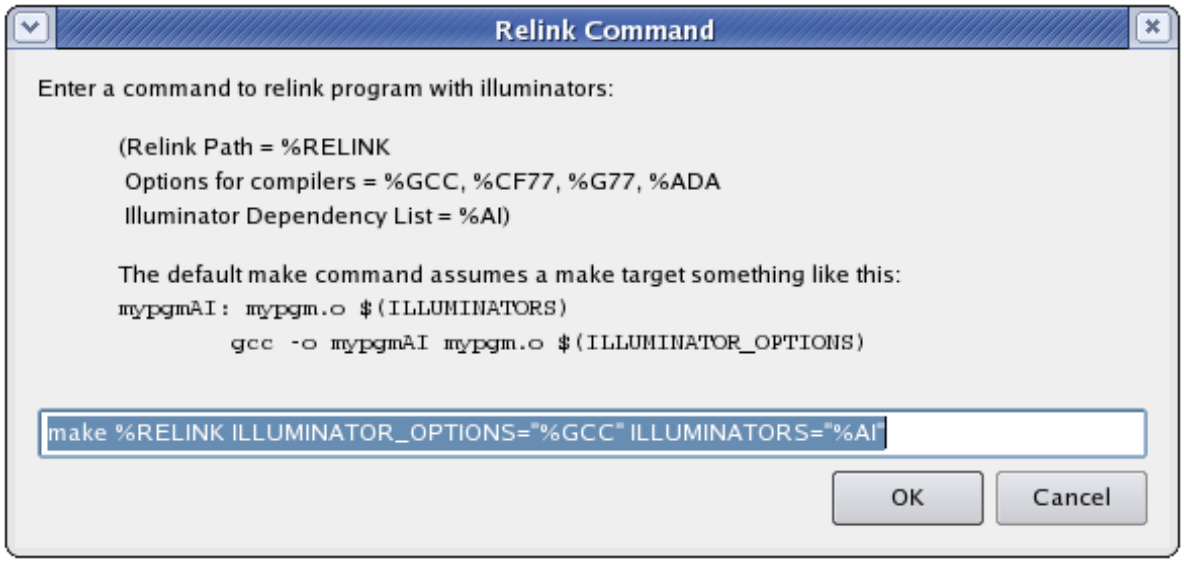


Figure 5-38. Edit Relink Command Dialog

- Right click on the Relink Command and select Edit Relink Command from the context menu: this also pops up a line editing dialog;

Figure 5-39. Relink Command Context Menu

- Right click on the tree item and select Set to Default make Command: this sets it to:

```
make %RELINK ILLUMINATOR_OPTIONS="%GCC" \
             ILLUMINATORS="%AI"
```

or,

- Right click on the tree item and select Set to Default a.link Command: this sets it to:

```
a.link -o %RELINK %ADA program
```

There are a number of substitution variables that may be specified in the Relink Command:

%RELINK

This is substituted with the Relink Path value. It is handy to use as a **`make`** target or as the operand of a **`-o`** option in **`a.link`**, **`gcc`**, or other compiler.

%AI

This is subsituted with the full paths of illuminators to be linked in. This is convenient for adding to a **`make`** file target's dependency list. The default **`make`** command passes this to **`make`** using the variable `ILLUMINATORS`.

%GCC, %G77, %CF77, %ADA

This is substituted with the options, files, and libraries that are needed to link with the illuminators using the **`gcc`**, **`g77`**, **`cf77`**, or **`a.link`** commands (respectively). This includes the NightTrace library. The default **`make`** command passes **`%GCC`** to **`make`** using the variable `ILLUMINATOR_OPTIONS`.

Illuminators

The Illuminators branch allows you to select which illuminators are linked into the program:

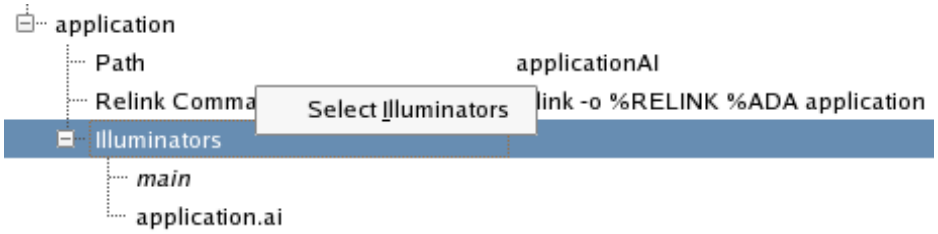


Figure 5-40. Illuminators Context Menu

By default the predefined `main` illuminator and any illuminator you create for the program are linked into it. You may select others:

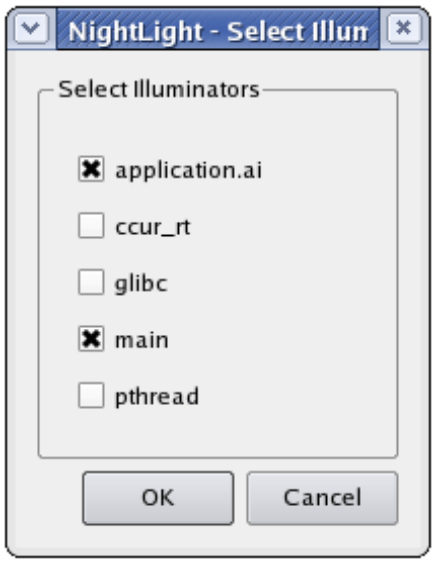


Figure 5-41. Select Illuminators Dialog

Remove illuminators from the list of illuminators linked in by unchecking them in the Select Illuminators Dialog, or by using the context menu to remove them:

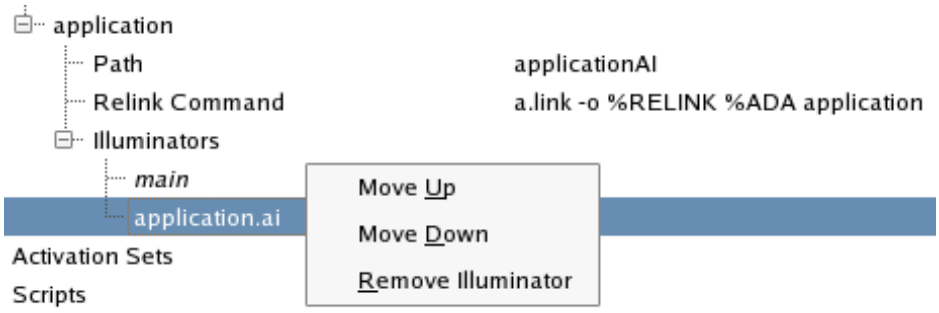


Figure 5-42. Relinked Illuminator Context Menu

Activation Sets

When illuminators are first linked with a program, they are inert. The fourth step in the work flow is to activate one or more of them. An *activation set* is a named set of activations that may be applied to your programs. During the course of analyzing a performance problem, you will typically turn on and off various illuminators, and adjust their options and detail levels. This can be done with one or more activation sets.

To create an activation set, right click on Activation Sets and select New Activation Set from the context menu that pops up:

Figure 5-43. Creating New Activation Set

You will be prompted for a name to give it. By default every program will be included in the set and every illuminator in the program will be activated. The below figure illustrates the default values given to settings on each illuminator:

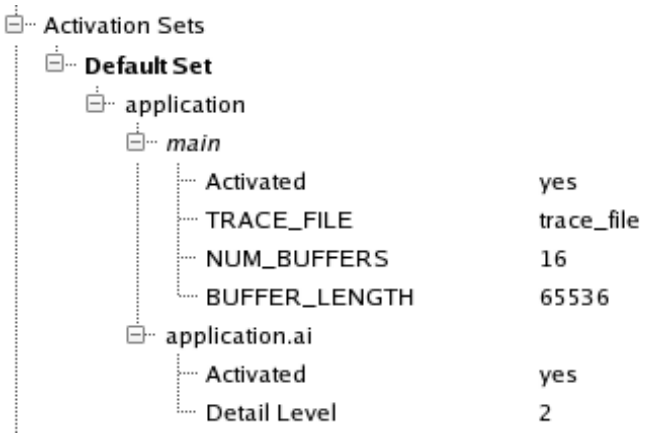


Figure 5-44. Default Options on Illuminators

To see a list of the current activations and options set for them, right click on Activation Sets and select Query Current Activations. You will get a dialog box showing the current activations:

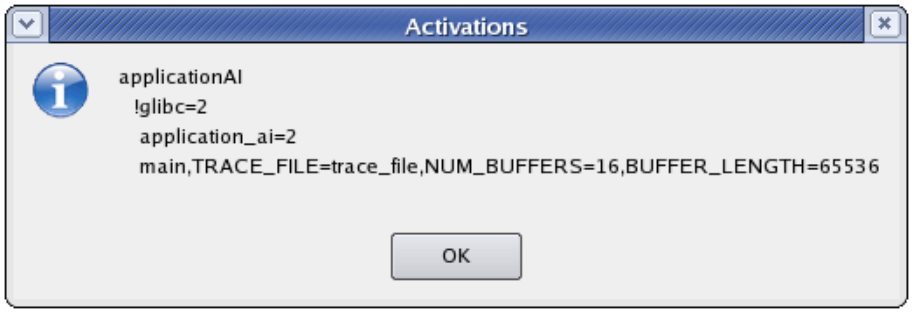


Figure 5-45. Query Current Activations Results

The "`!`" preceeding an illuminator name indicates it is not activated. Periods in illuminator names are transformed to an underscore since internally these are parts of C symbol names.

Settings For "main" Illuminator

The main illuminator is special. It does not generate any trace events, but it does do a `trace_begin()` call before `main()` begins executing. The settings allow you to specify parameters to pass to that `trace_begin()`.

Activated

Controls whether the illuminator will be activated (yes) or deactivated (no). It defaults to yes.

TRACE_FILE

Sets the path to the file that the NightTrace events will be recorded in. It defaults to "**`trace_file`**" in the current working directory.

NUM_BUFFERS

Sets the number of buffers that the NightTrace library will use for recording events. It defaults to 16 buffers.

BUFFER_LENGTH

Sets the number of bytes in each buffer that the NightTrace library will use for recording events. It defaults to 65536 bytes.

Settings For Ordinary Illuminators

All other illuminators record events. By default, there are three detail levels for each illuminator, named `1`, `2`, and `3`. Each record more detail than the next lower numbered one. Customized illuminators may have additional detail levels, whose names are not limited to numbers, but may be anything.

Activated

Controls whether the illuminator will be activated (`yes`) or deactivated (`no`). It defaults to `yes`.

Detail Level

Sets the name of the detail level that the activated illuminator will use to record events. It defaults to `2`.

Context Menu for an Illuminator

Right clicking on an illuminator in the `Activation Sets` section brings up a context menu:

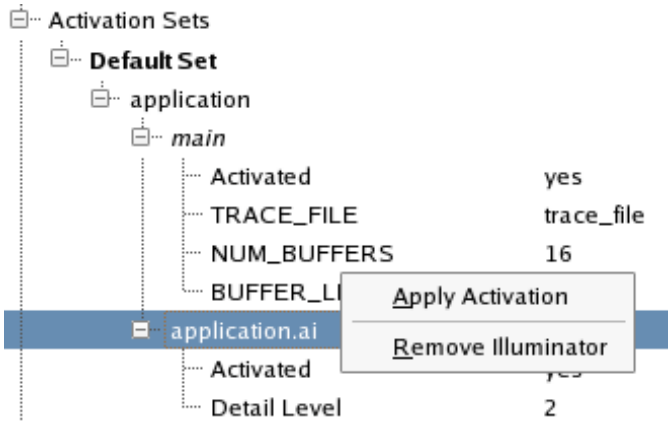


Figure 5-46. Context Menu on an Illuminator in an Activation Set

Apply Activation

Activates (or deactivates) a single illuminator in a single program. The other illuminators and other programs are not affected.

Remove Illuminator

Removes an illuminator from the activation set. Once removed from the activation set, that activation set will neither activate nor deactivate that illuminator. The illuminator remains linked with the program in whatever activation state it already has. For example, you can set up a collection of activation sets that control the activation

of the `glibc` illuminator and another collection of activation sets that control all illuminators but `glibc`.

Context Menu for a Program

Right clicking on a program in the Activation Sets section brings up a context menu:

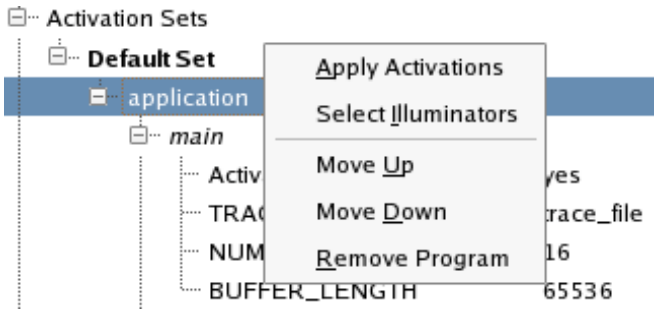


Figure 5-47. Context Menu on a Program in an Activation Set

Apply Activations

Applies the activations to just this program's illuminators, but does not modify any other program.

Select Illuminators

Brings up a dialog that allows selecting which illuminators that are linked in to this program are to be activated (or deactivated). Deselected illuminators will remain linked with the program in whatever activation state they are already in.

Figure 5-48. Select Illuminators Dialog

Move Up,
Move Down

Rearranges the order of the programs. This is purely cosmetic.

Remove Program

Removes the program from this activation set. No changes to the activations of the program will be made by this activation set.

Context Menu for an Activation Set

Right clicking an activation set name in the Activation Sets section brings up a context menu:

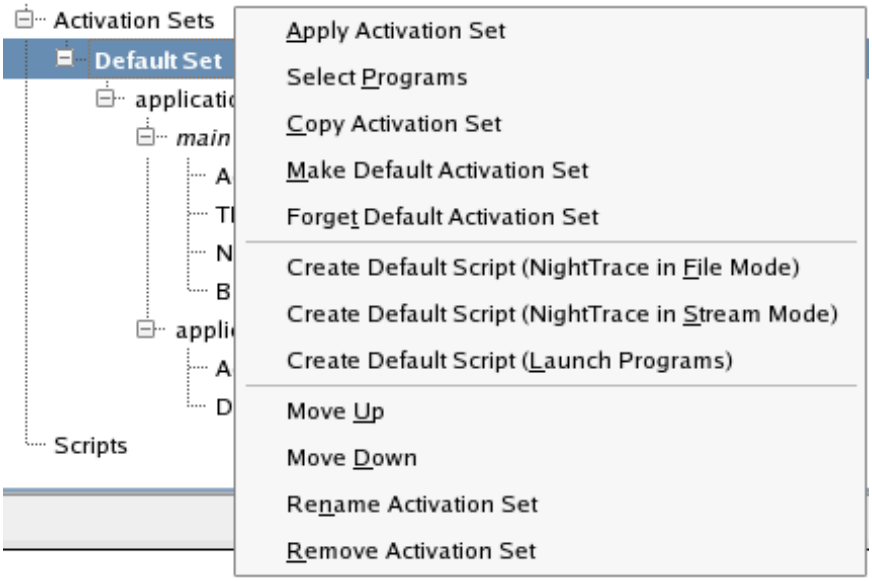


Figure 5-49. Context Menu on an Activation Set

Apply Activation Set

Applies all the activations (or deactivations) of all the illuminators in all the programs that are in this activation set.

Select Programs

Brings up a dialog that allows you to choose which programs are in this activation set. By default, all programs are selected.

Copy Activation Set

Brings up a dialog asking for the name of a new activation set, and creates a copy of this activation set with that name.

Make Default Activation Set

Designates a single activation set as the default activation set. It immediately applies it. Then, whenever a program gets relinked, this activation set is immediately applied to it.

Whenever a change is made to a setting in the default activation set, that change is immediately applied to that illuminator in that program. This means if you apply a different activation set, then modify the default activation set, the current activations will be a mixture of the two activation sets.

The Wizard window (or page) will modify the default activation set. If there isn't one, it will create one called Wizard.

The default activation set is displayed in a bold font.

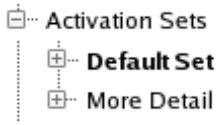


Figure 5-50. Default Activation Set in Bold

Forget Default Activation Set

Removes the default activation set designation. It does not change the current activations, but whenever a program is relinked in the future, it will default to having no illuminators activated.

Create Default Script (NightTrace in File Mode), Create Default Script (NightTrace in Stream Mode), Create Default Script (Launch Programs)

Creates scripts based on the programs and settings in an activation set. See "New Script from Activation Set (NightTrace in File Mode)" on page 5-59 and "New Script from Activation Set (NightTrace in Stream Mode), New Script from Activation Set (Launch Programs)" on page 5-60 for details on these actions.

Move Up, Move Down

Changes the order of the activation sets. This is purely cosmetic.

Rename Activation Set

Prompts you for a new name for an activation set.

Remove Activation Set

Deletes an activation set from the session. No changes are made to the current activations.

Scripts

The fifth and final step in the workflow is to run your instrumented programs and NightTrace. The Scripts section of NightLight allows you to set up scripts for automating this process. Right click on Scripts to get a context menu for creating a script:

Figure 5-51. Scripts Context Menu

New Script

Selecting New Script prompts for a name for the script. The script has some settings for controlling how it is invoked. When NightLight invokes a script, it places it in a file called **`.nlight_script`***n* in the current working directory. By default, output from the script is directed to the Console window (or page). If it is necessary to interact with the script, you can run it under an X Terminal, Gnome Terminal, KDE Terminal, or invoke it by a custom method.

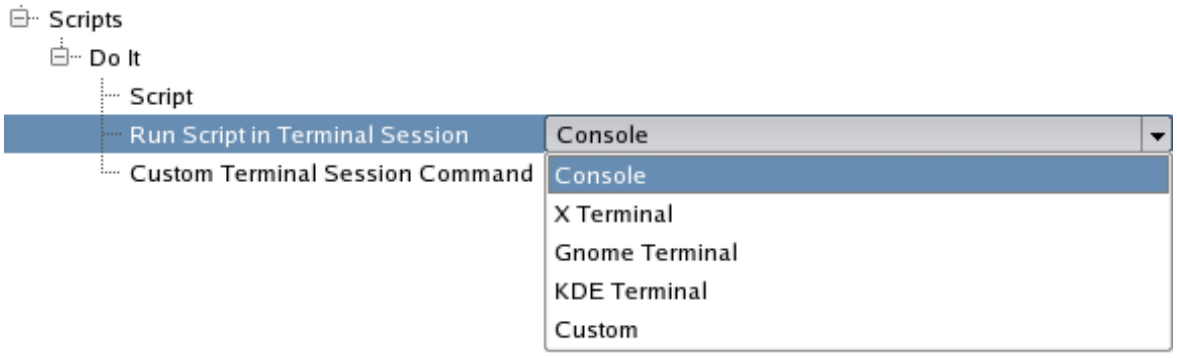


Figure 5-52. Run Script in Terminal Session

When Custom is selected, the script is invoked by the Custom Terminal Session Command. By default, this is an **`xterm`** command for illustration purposes. The Custom Terminal Session Command can be used to invoke the script any arbitrary way or to pass an option to the script:

Figure 5-53. Invoking a Script on the Console While Passing an Option

Double click on Script or select Edit Script from its context menu to bring up an editor dialog for editing the script:

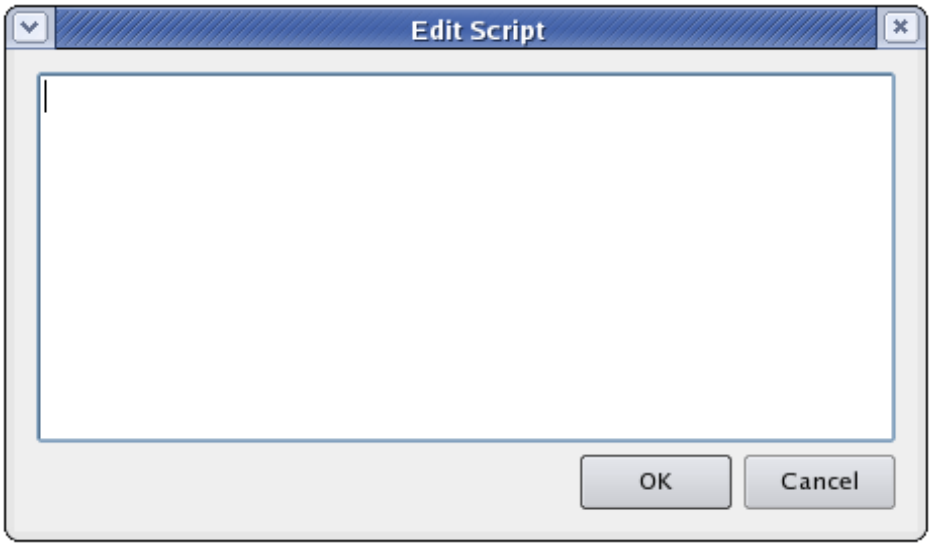


Figure 5-54. Edit Script Dialog

New Script from Activation Set (NightTrace in File Mode)

As a convenience, NightLight can use the information in an activation set to create a first draft of a script. NightLight prompts you for an activation set and then scans the "`main`" illuminators for trace files and create commands to start a user daemon for each one, run each of the programs in succession, stop the daemons, and invoke NightTrace. As a shortcut, you can also right click on an activation set and select Create Default Script (NightTrace in File Mode) from the context menu (see "Create Default Script (NightTrace in File Mode), Create Default Script (NightTrace in Stream Mode), Create Default Script (Launch Programs)" on page 5-57). You can then edit the script to add options to the commands, control the order your programs are run, etc:

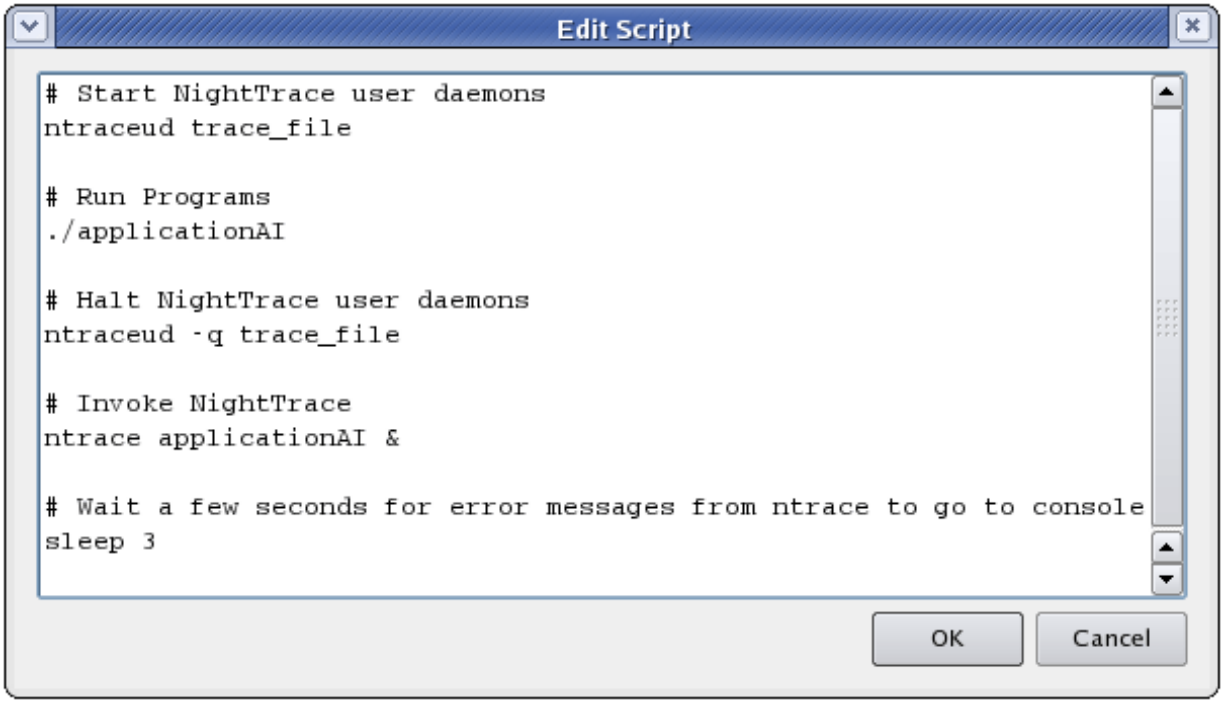


Figure 5-55. Default Script created for an Activation Set (NightTrace in File Mode)

New Script from Activation Set (NightTrace in Stream Mode), New Script from Activation Set (Launch Programs)

If you want to use NightTrace in its streaming mode, you can create a pair of scripts: one to launch NightTrace with the appropriate daemons, and one to launch your instrumented programs. The New Script from Activation Set (NightTrace in Stream Mode) and New Script from Activation Set (Launch Programs) will prompt you for an activation set and create a first draft of the scripts for doing this. As a short cut, you can also right click on an activation set and select the corresponding Create Default Script context menu item (see "Create Default Script (NightTrace in File Mode), Create Default Script (NightTrace in Stream Mode), Create Default Script (Launch Programs)" on page 5-57).

Here are the default scripts generated for a more elaborate example:

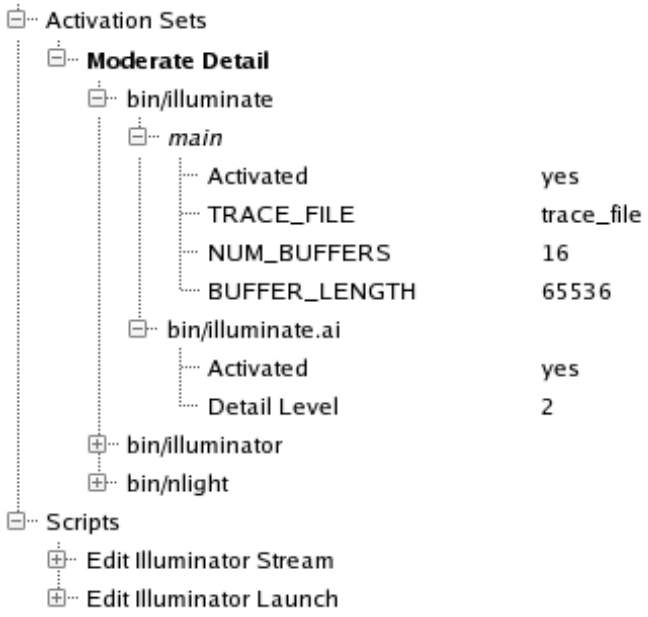


Figure 5-56. Activation Set for an Elaborate Script Example

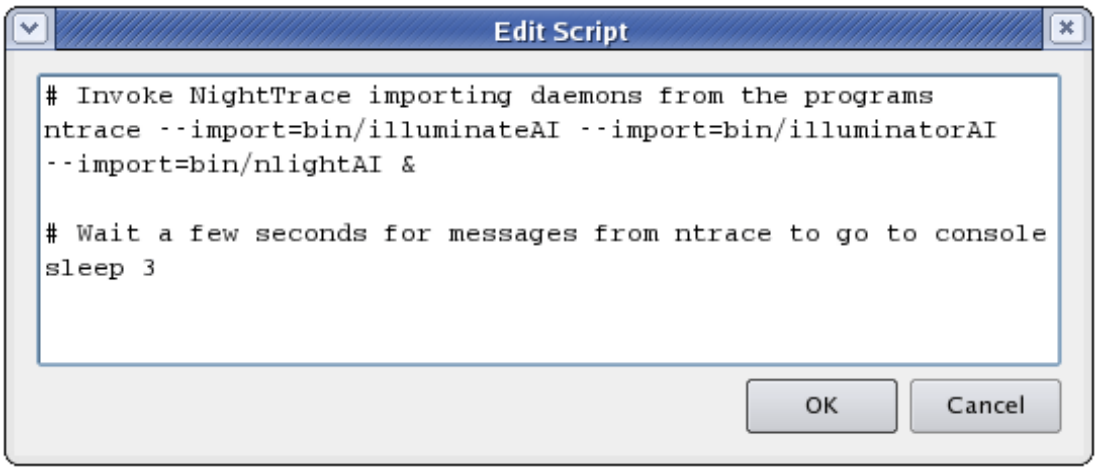


Figure 5-57. Script for Invoking NightTrace in Stream Mode

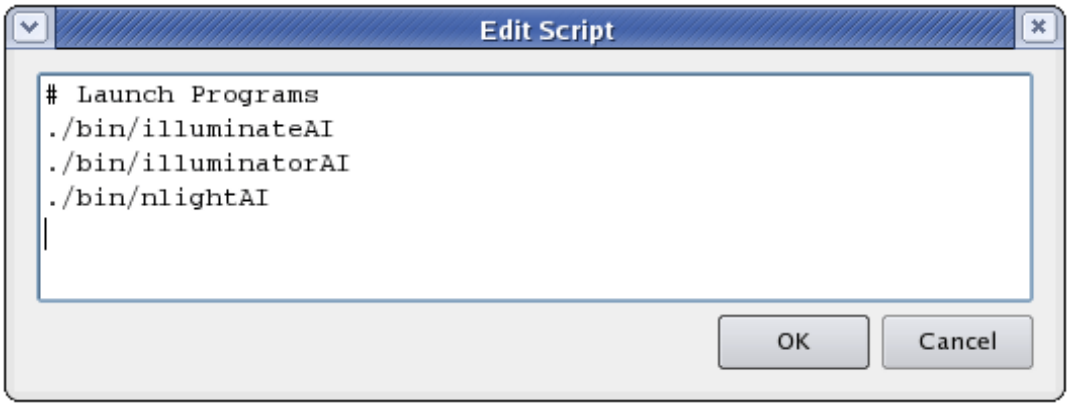


Figure 5-58. Script Launching Instrumented Programs

Now, in actuality, these programs need to be given parameters, and I may not be interested in running them all for debugging a particular problem. I might also need to do some setup for them. So I'm going to modify the script like this:

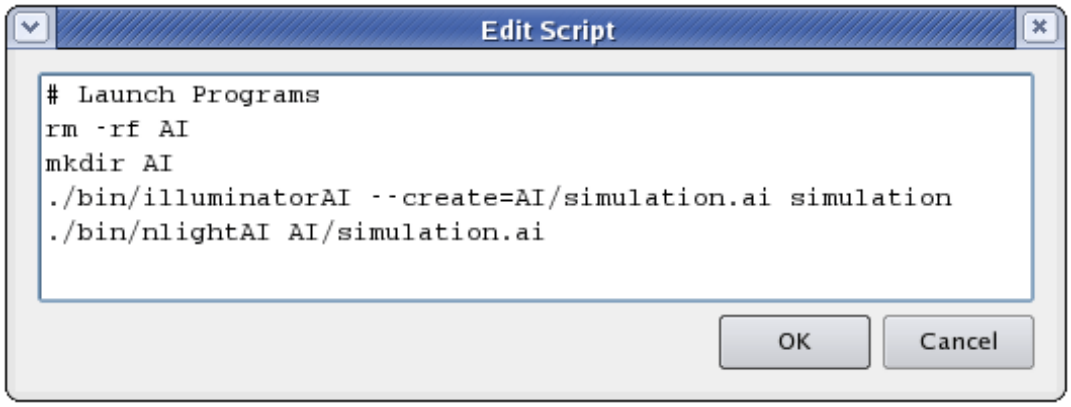


Figure 5-59. Modified Script Launching Instrumented Programs

Console

The Console window (or page) captures output from external commands and scripts. Each command invoked will have a heading with a time stamp, description of the action, and command being invoked. The command output will come next, then finally the status returned by the command. Scripts can have their output directed to a different terminal session.

The status will be green if zero, and red if non-zero. You will also get a warning dialog for any non-zero status that is returned. Scripts might not return an error status but still have error messages.

Figure 5-60. Non-zero Status Warning

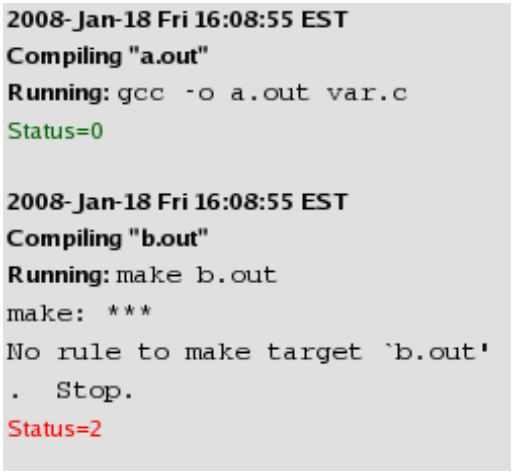


Figure 5-61. Console Output

Predefined Illuminators

Detail Levels

Except for `main`, all predefined illuminators have the three default detail levels: `1`, `2`, and `3`. The table below details what information is recorded on the events that NightLight generates for function entry and return events.

Table 5-1. Values Recorded As Arguments to Illumination Events

	1	2	3
return address (entry events)	x	x	x
frame pointer (entry events)		x	x
byte limit on aggregate size (all events)	16	16	16
parameters (entry events)		x	x
indirect through pointer parameters (entry events)			x
return values (return events)	x	x	x
indirect through pointer return values (return events)			x
errno (return events)			x

main

The `main` illuminator is special. It does not record any events. Linking with and activating it causes `trace_begin()` to be called before `main()` is called. This is necessary if the traced program does not do its own `trace_begin()` call.

If a program does its own `trace_begin()` call, do not use this illuminator. In this situation, NightLight and NightTrace will not know automatically what user trace daemon is needed by the instrumented program, so generated scripts will have to be edited to include the appropriate daemon and trace file.

glibc

The `glibc` illuminator illuminates functions from the system C library. The thousands of functions are partitioned into dozens of named groups for convenience when customizing the `glibc` illuminator (see "Groups" on page 5-90). Use the Editor window (or page) by editing the illuminator, select the Report menu item from the context menu from the Create, Customize, and Build Illuminators section of the session manager, or use the following command to see a list of all groups and their functions:

```
nlight --report=glibc
```

pthread

The `pthread` illuminator illuminates functions from the system `pthread` library. The functions are partitioned into two named groups for convenience when customizing the `pthread` illumintor:

- `glibc` - functions that are redundant with functions in the C library; and
- `pthread` - the functions implementing threads.

ccur_rt

The `ccur_rt` illuminator illuminates functions from the `ccur_rt` library. This illuminator will be present only on systems with the `ccur_rt` library installed. The numerous functions are partitioned into several named groups for convenience when customizing the `ccur_rt` illuminator. Use the Editor window (or page), use the Report context menu item, or use the following command to see a list of all groups and their functions:

```
nlight --report=ccur_rt
```

Illuminator Files

The following files are created in the *illuminator* directory:

`config.xml`

The file that holds all the settings and customizations for an illuminator. An unbuilt illuminator will contain only this file.

`next_event.txt`

The next event number after the last one assigned. Its purpose is to assist in creating multiple wrapper libraries that use contiguous ranges of events.

```
$ nlight --build=fred --event_ids=1000-2000
$ nlight --build=barney \
         --event_ids=`cat fred/next_event.txt`-2000
```

illuminator`.h`

Header file that `#defines` a name for each event for use in calling the NightTrace analysis API. The names are of the form:

`TRACE_EVENT_`*illuminator*`_ENTER_`*function* and
`TRACE_EVENT_`*illuminator*`_RETURN_`*function*.

When a function has been aliased to have multiple names (usually a strongly and a weakly defined name), only a single event pair is allocated for it. The function name used to build the event name is the shortest alias (then lexically earliest if there are two or more shortest aliases). Each alias will get its own wrapper function, but they will each record the same entry and return event IDs.

illuminator`.map`

NightTrace event map naming the events. The names are of the form:

`ENTER_`*function* and
`RETURN_`*function*.

illuminator_level`.fmt`

NightTrace format table called `illumination`. There is one for each detail level so NightTrace knows what details were recorded in the trace file.

illuminator_level`.o`

Object file that gets copied into the user program by **`nlight --illuminate`** to control the level of detail recorded by each function in the wrapper library.

illuminator_level`.list`

The list of functions to wrap or not wrap for each detail level. It is used by the **`nlight --illuminate`** command.

illuminator`.o`

Relocatable object file containing all the “wrapper” functions.

illuminator`.vararg`

NightTrace table called `vararg_functions` indexed by entry event number. The indexed entry will be “`true`” if the corresponding function is a “vararg” function (and thus doesn’t generate a return event) or “`false`” otherwise.

NightLight Command Line Mode

Illuminators can be created, manipulated, used, activated, and deactivated by using **`nlight`** in command-line mode rather that running the tool in GUI mode.

Commands for Manipulating an Illuminator

nlight --create

Usage:

```
$ nlight --create=illuminator [options] [object files]
```

Creates a directory called *illuminator* (with periods changed to underscores) and places in it a **`config.xml`** file that reflects the options and object files specified on the remainder of the command line. If *illuminator* already exists, it will be modified to include the additional *options* and *object files* that are specified.

The following options may be specified:

`--aggregate_limit=`*limit*
`--config=`*config.xml*
`--do_nodebug`
`--dont_nodebug`
`--event_ids=`*N-[M]*
`--install=`*path*
`--iunderscores`
`--iregex=`*regex*
`--istd`
`--xunderscores`
`--xregex=`*regex*
`--xstd`

The *object files* that may be specified are those containing the functions to be illuminated. They may be a whole program, archives, shared objects, individual object files, or debug-info files. If the DWARF debug information has been placed in a separate debug-info file, it must be listed immediately after its corresponding object file.

`--aggregate_limit=`*limit*

Limits the recording of aggregate values to *limit* bytes. Aggregates might get recorded with an event if a function's parameter or return value is a C/C++ `struct` type, for example. Only the first *limit* bytes of the aggregate are recorded.

This option may also be set in a *config.xml* file:

```
<defaults><options aggregate_limit=limit/></defaults>
```

(See "`aggregate_limit=`limit" on page 5-106).

The limit must be at least 16 bytes. The default limit is 16 bytes.

--config=*config.xml*

Reads configuration from an XML file. More than one instance of this option may be specified to merge several such files together. Options specified on the command line after the **--config** option will override options set in the *config.xml* file. One use of this might be to generate a customized `glibc` illuminator.

```
$ nlight --create=myglibc \
       --config=/usr/lib/NightTrace/illuminators/glibc/config.xml \
       --aggregate_limit=64
```

This would initialize **myglibc/config.xml** with **/usr/lib/NightTrace/illuminators/glibc/config.xml**, but change the aggregate limit from 16 to 64.

--do_nodebug, --dont_nodebug

Creates or blocks creation of trace events for functions that have no DWARF debug information. The default is to not create such trace events. Only entry events are generated for functions without debug information. An alternative to **--do_nodebug** is to use a *config.xml* file to provide a signature for the function (See "declare" on page 5-102).

This option may also be set in a *config.xml* file:

```
<defaults><options nodebug={yes|no}></defaults>
```

(See "nodebug={yes|no}" on page 5-108).

--event_ids=*N-[M]*

Specifies the range of NightTrace event IDs to use for the function entry and return events. If the range is exceeded, a warning is generated.

This option may also be set in a *config.xml* file:

```
<defaults><options event_ids=N-[M]></defaults>
```

(See "event_ids="N-[M]"" on page 5-108).

The defaults for *N* and *M* are 12000 and 32767 respectively. The highest possible event ID is 32767.

If the upper bound is 32767 and the illuminator is built through the graphical interface, NightLight will change the lower bound to be a value in the range 12000 through 32700 so that the illuminator's event range will not overlap other illuminators in the session that also have their upper bound set to 32767.

--install=*path*

Specifies an installed location for an illuminator, in contrast to the location where it is actually built. This path is recorded in the object files for **ntrace** to find the

event map and format tables (see "Using NightTrace with Illuminators" on page 5-75).

`--i*, --x*`

Includes or excludes functions from getting entry and return events based on the functions' names. Multiple instances of these options may be specified. The last one specified that matches a function's name determines whether that function is included or excluded. Excluded functions are not included in the **`--populate`** output.

`--iunderscores, --xunderscores`

Includes or excludes functions whose names start with an underscore character. All aliases of a function and the fully qualified C++ name (if applicable) must begin with an underscore in order to match these options (in contrast to **`--iregex=_.*`** or **`--xregex=_.*`**). A fully qualified C++ name matches if the function name or name of any containing classes start with an underscore.

The rationale for this is that functions and class names that begin with underscores are typically vendor implementation routines that are of less interest. But it is also common practice to create a strongly defined function that starts with an underscore, then weakly define aliases to that function that do not. These functions, like many in Glibc (see NOTE), are likely to be interesting, and so aren't matched by these options.

NOTE

Many functions in Glibc for which all aliases begin with an underscore do not follow standard function call conventions, and so should never be traced via Application Illumination.

These options may also be specified in a *config.xml* file:

```
<defaults><options underscores={yes|no}/></defaults>
```

(See "`underscores={yes|no}`" on page 5-108).

The default is **`--xunderscores`**.

`--iregex=`*regex*`, --xregex=`***regex***

Includes or excludes functions whose names match a POSIX regular expression (see **`regex(7)`**). A function name matches the regular expression if any alias or fully qualifed C++ name (if applicable) matches it. The regular expresssion must match the whole name (an implicit `^` and `$` is placed before and after the regular expression respectively).

These options may also be specified in a *config.xml* file:

```
<defaults>
   <option iregex=regex/>
   <option xregex=regex/>
</defaults>
```

(See "`iregex="regex", xregex="regex"`" on page 5-109).

By default

```
main,
.*\.internal_io.ada, and
.*\.internal_io\.ada\.\..*
```

are excluded.

To include only functions matching a particular *regex*, first exclude all functions:

```
--xregex=.* --iregex=regex
```

`--istd, --xstd`

Includes or excludes C++ functions in the `std` namespace.

These options may also be specified in a *config.xml* file:

```
<defaults><option std={yes|no}/></defaults>
```

(See "`std={yes|no}`" on page 5-109).

The default is to exclude C++ functions in the `std` namespace. Such functions are often inlined and so tracing them usually doesn't provide a lot of useful information.

nlight --populate

Usage:

```
$ nlight --populate=illuminator [options] [object files]
```

Creates or updates (like **`--create`**) the *illuminator*'s **`config.xml`** file to add the *options* and *object files* specified, then populates the **`config.xml`** file with a list of all the functions found on the *object files* that it will generate trace points for and all the global variables it can record as arguments to return events. This can be a great convenience when you want to create a number of function-specific customizations by editing the **`config.xml`** file. If such customizations are made, they will be retained if you run the **`nlight --populate`** command again, which you will likely want to do anytime you add or remove functions or change the function's signatures that you are illuminating.

nlight --build

Usage:

$ **nlight --build=***illuminator [options] [object files]*

Creates or updates (like **--create**) the *illuminator*'s **config.xml** file to reflect the *options* and *object files* specified, then builds the "wrapper" functions, event map, format tables, etc. You will want to do this any time you change the types or function signatures that Application Illumination uses to create trace points.

By default, three detail levels are created for the illuminator: `1`, `2`, and `3`. You may edit the **config.xml** file to modify these detail levels or to create custom detail levels.

nlight --report

Usage:

$ **nlight --report=***illuminator*

Generates a report about an *illuminator* on functions, function groups, global variables, etc. For example:

```
$ nlight --report=pthread
The following global variables were found:
The following subroutines had no debug information or
<declare>:
   __pread64
   __pwrite64
   lseek64
   pread
   pread64
   pwrite
   pwrite64
The following subroutines were excluded because of their
names:
   __errno_location
   __h_errno_location
   __libc_allocate_rtsig
   …
   _pthread_cleanup_pop
   _pthread_cleanup_pop_restore
   _pthread_cleanup_push
   _pthread_cleanup_push_defer
The following subroutines are in group "glibc":
   _IO_flockfile
   _IO_ftrylockfile
   _IO_funlockfile
   …
   wait
   waitpid
   write
The following subroutines are in group "pthread":
```

```
   __pthread_atfork
   __pthread_getspecific
   …
   pthread_testcancel
   pthread_timedjoin_np
   pthread_tryjoin_np
   pthread_yield
   sem_close
   sem_destroy
   sem_getvalue
   sem_init
   sem_open
   sem_post
   sem_timedwait
   sem_trywait
   sem_unlink
   sem_wait
The following subroutines are in no group:
$
```

Commands for Linking with Illuminators

Once built, an illuminator's "wrapper" functions must be linked into your program with the **`-Wl,--emit-relocs`** and either **`-lntrace`** or **`-lntrace_thr`** options. The **`nlight`** program with the below options can be used between back-quotes to conveniently generate all the options to reference the needed object files and options. When an illuminator is specified with a relative path, the program will search for it first relative to the current directory, and then relative to **`/usr/lib/NightTrace/illuminators`**. Alternatively, an absolute path to the illuminator directory may be given.

When an illuminator is first linked into your program, it is inert. It does not intercept any function calls or interfere with your program's performance at all until it is activated with the **`nlight --illuminate`** command (see "Command for Activating and Deactivating Illuminators" on page 5-74).

nlight --gcc

Usage:

```
$ gcc ... `nlight --gcc [-t] illuminator_list`
```

Generates options suitable for **`gcc`** to link in a list (separated by whitespace) of illuminators. The **`-t`** option specifies the use of the threaded **`ntrace`** library.

This generates the following options:

- *illuminator_path*`/`*illuminator*`.o` (for each illuminator)
- **`-Wl,--emit-relocs`**
- **`-lntrace`***`[_thr]`*

nlight --g77

Generates options suitable for **`g77`**. See "nlight --g77" on page 5-74.

nlight --cf77

Generates options suitable for **`cf77`**. See "nlight --cf77" on page 5-74.

nlight --ada

Generates options suiltable for **`a.link`**. See "nlight --ada" on page 5-74.

This generates the following options:

- **`-ld`** *illuminator_path*`/`*illuminator*`.o` (for each illuminator)
- **`--emit-relocs`**
- **`-so=ntrace`***`[_thr]`*

Command for Activating and Deactivating Illuminators

Once the illuminators are linked into a program, they can be activated by using the **`nlight --illuminate`** command. This command scans the user program for calls to the functions to be traced, and redirects them to the "wrapper" functions in the illuminator that record the entry event, call the real function, record the return event, and return.

Usage:

```
$ nlight --illuminate program [[!]main[,options]] \
                      [[!]illuminator[=level]]...
```

program

Specifies the program you linked with illuminators. `nlight --illuminate` may be run on the program multiple times to turn on and off various illuminators and to change their detail levels.

`!`

Deactivates the illuminator the "`!`" is prefixed to. When deactivated, an illuminator has no run-time overhead.

`main`[`,`*options*]

Specifies the `main` illuminator and its options. This illuminator is special. It "wraps" only the `main()` routine, and records no events. Instead, it performs a `trace_begin()` call (see "trace_begin, Trace.begin" on page 2-7). Rather specifying a

detail level, you may specify a comma-separated list of options to the `trace_begin()` call:

- **`TRACE_FILE=`***filename*

  Specifies the name of the file that will hold the trace events. The default is **`trace_file`**.

- **`NUM_BUFFERS=`***count*

  Specifies the number of buffers used for recording trace events. The default is 8.

- **`BUFFER_LENGTH=`***size*

  Specifies the length in bytes of each buffer used for recording trace events. The default is 32768.

illuminator

Specifies the name of the illuminator. This can be an absolute or relative path to the directory containing the illuminator's files. Relative paths will be searched for relative to the current directory and then relative to **`/usr/lib/NightTrace/illuminators`**. The following illuminators are provided in **`/usr/lib/NightTrace/illuminators`**:

- **`main`**
- **`glibc`**
- **`pthread`**
- **`ccur_rt`**

level

Specify the level of detail to be recorded by the illuminator's events. The default is `2`. By default, illuminators have detail levels `1`, `2`, and `3`. These levels may be customized, or custom details may be created, for any illuminator.

Using NightTrace with Illuminators

Illuminators have a NightTrace event map and, for each detail, a NightTrace format table, within them. The absolute path to these files are embedded in programs that have the illuminator linked in. If the `main` illuminator is used, the (possibly relative) path to the trace file is also embedded in the program. You may specify a program on the `ntrace` command line, and NightTrace will extract these embedded paths and use them.

Usage:

```
$ ntrace a.outAI
```

Note that because the path to the trace file may be a relative path, the **`ntrace`** command should be run with the current working directory being the same as when *a.outAI* was run.

Customizing an Illuminator with the Editor

The Editor is invoked by selecting the Edit context menu item on an illuminator in the Illuminators section of the session manager window (or page), by double clicking on that same illuminator, or by specifying an illuminator on the **`nlight`** command line. The editor can be in its own window or in a page in the session manager, depending on the Show Editor in Page setting in the View menu.

The Editor presents the configuration of an illuminator in a tree structure much like the session manager.

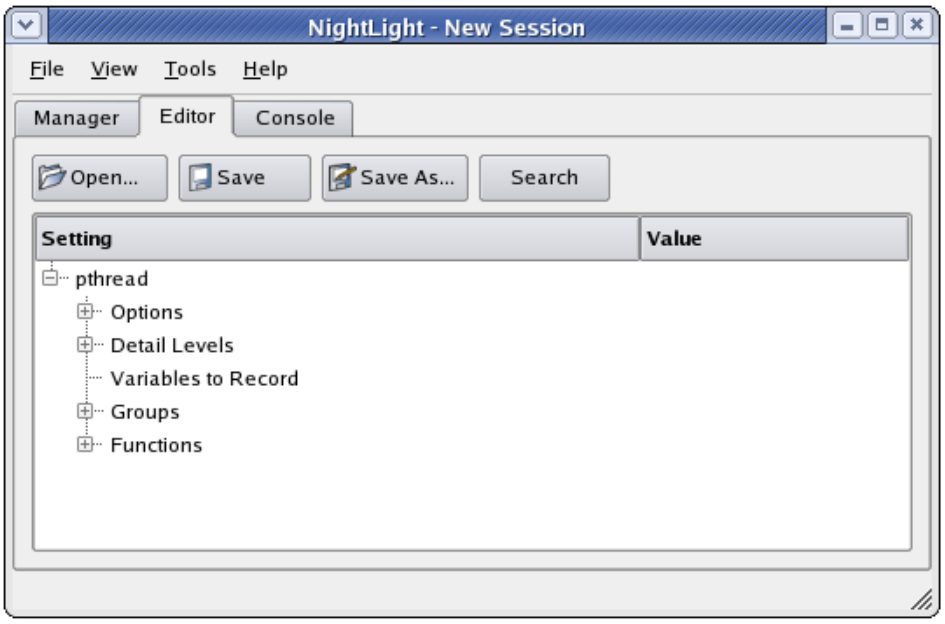


Figure 5-62. Editor Page

Buttons

The row of buttons above the settings tree allow loading and saving of the **`config.xml`** files that define the customizations of an illuminator and toggle the search bar.

Open...

Launches a standard file selection dialog which allows you to specify an illuminator's **`config.xml`** file to edit.

If changes have been made to the current illuminator but have not yet been saved, NightLight will ask you if you wish to save the current illuminator before proceeding.

Save

Saves the current illuminator to a **`config.xml`** file quickly.

You are not prompted for the filename where the illuminator is to be saved. It is automatically saved to the same file it was opened from or previously saved to.

Save As...

Launches a standard file selection dialog which allows you to specify the filename where the illuminator's **`config.xml`** file will be saved.

Search

Toggles displaying the search bar at the bottom of the Editor window (or page).

Search Editor

The Search Editor menu item in the View menu or the Search button turns on the search bar in the Editor window (or page). All of the items in the editor's trees also include Search in their context menus.

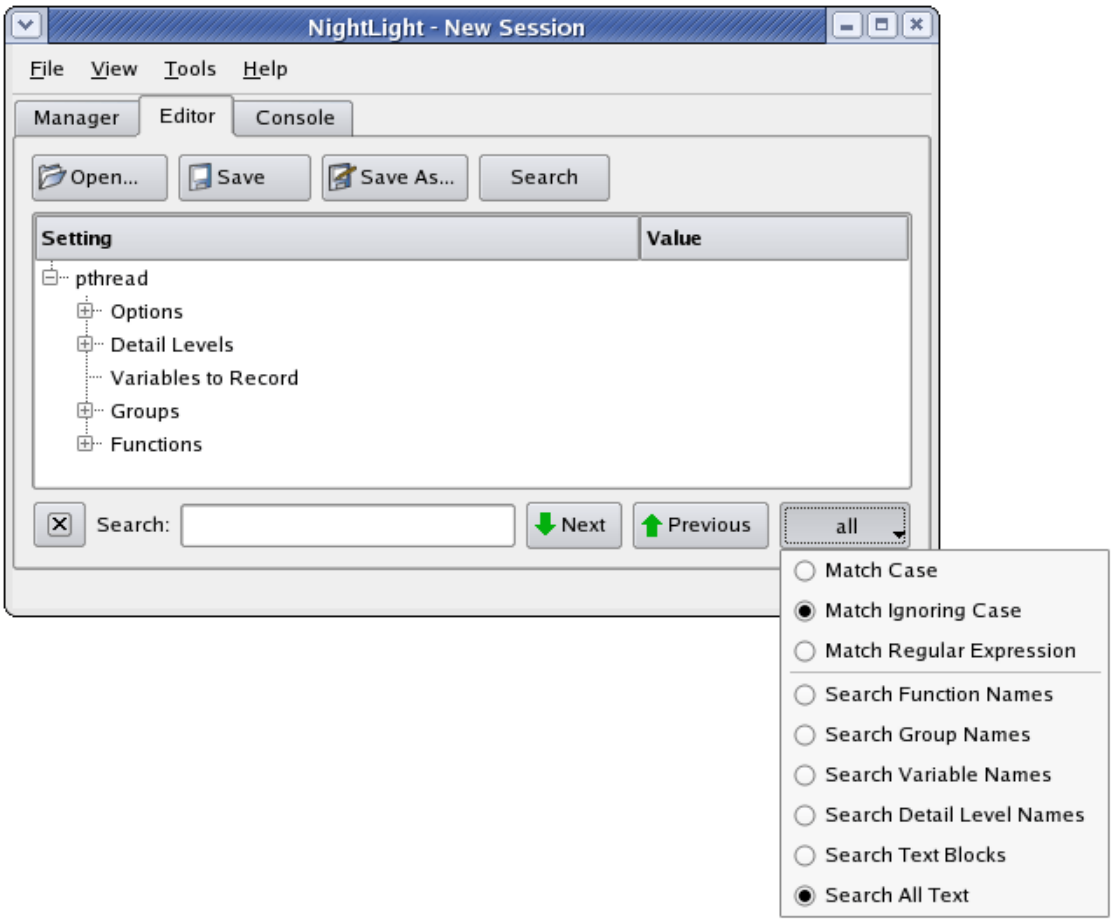


Figure 5-63. The Search Bar

Closes the search bar.

Specifies the text or regular expression to search for. If a Search context menu is used, it is initialized with a regular expression that will match the value of the item that was right clicked on exactly.

Searches for the next or previous instance of the search string or regular expression.

Opens up a pull down menu that allows you to specify search options. The label on the button reflects the settings of these search options.

Capitalization and Punctuation indicate case and regular expression settings.

All

Capitalized: matches case.

all

All lower case: matches ignoring case.

All*

Capitalized with an asterisk: matches regular expression.

The word indicates the type of value to search for. When using the Search context menu item on a setting with a particular type, the type setting will be set to that type, regular expression.

All, all, All*

Searches all text in the tree structure, including values that are multi-line blocks of text.

Group, group, Group*

Searches only group names.

Function, function, Function*

Searches only function names

Level, level, Level*

Searches only detail level names.

Text, text, Text*

Searches only multi-line text block values.

Variable, variable, Variable*

Searches only variable names.

Options

The options section contains settings that are not specific to any detail level, group, or function.

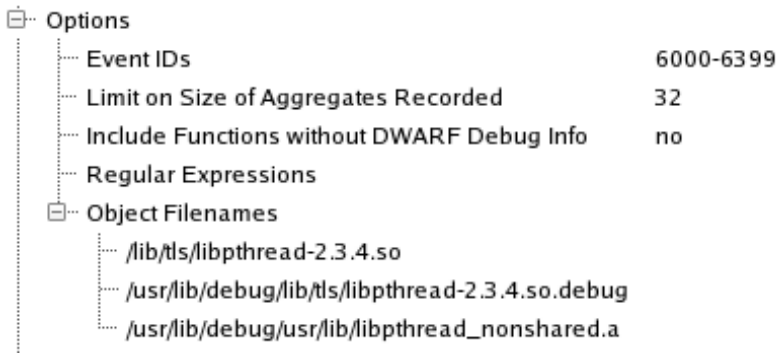


Figure 5-64. Options

Event IDs

Specifies the range of event_ids to be mapped to entry and return events. (See "`event_ids="N-[M]"`" on page 5-108). If the upper bound of the specified range is 32767, NightLight will take over assigning the lower bound to the range 12000 through 32700 such that the assigned event IDs won't overlap other managed illuminator's ranges in the session.

Limit on Size of Aggregates Recorded

Limits the number of bytes of an aggregate that may be recordeed with an event. The limit must be at least 16 bytes. (See "`aggregate_limit="limit"`" on page 5-108).

Include Functions without Dwarf Debug Info

Specifies whether functions that have no debug information are to be illuminated or not. Return events are not generated for functions without debug information. (See "`node-bug={yes|no}`" on page 5-108).

Regular Expressions

Specifies whether function names that match regular expressions are to be illuminated or not. Multiple expressions may be specified. Each regular expression either specifies functions to include in the illuminated functions list or specifies functions to exclude. If more than one regular expression matches a function name, the last one to match overrides the previous ones. Right click on Regular Expressions to bring up the context menu of regular expressions that may be added to the list.

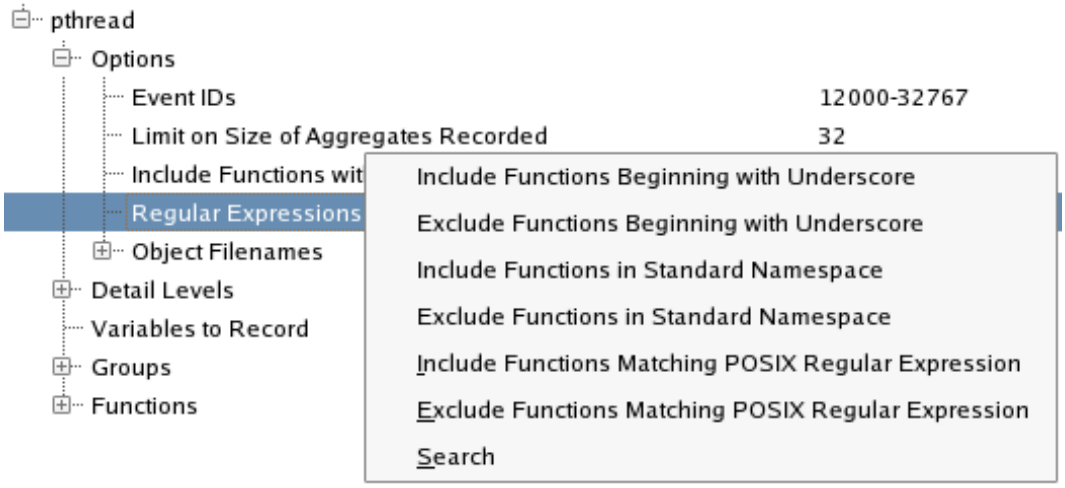


Figure 5-65. Regular Expressions Context Menu

Include Functions Beginning with Underscore, Exclude Functions Beginning with Underscore

Includes or excludes functions whose names start with an underscore character. All aliases of a function and the fully qualified C++ name (if applicable) must begin with an underscore in order to match these options (in contrast to **`--iregex=_.*`** or **`--xregex=_.*`**). A fully qualified C++ name matches if the function name or name of any containing classes start with an underscore.

The rationale for this is that functions and class names that begin with underscores are typically vendor implementation routines that are of less interest. But it is also common practice to create a strongly defined function that starts with an underscore, then weakly define aliases to that function that do not. These functions, like many in Glibc (see NOTE), are likely to be interesting, and so aren't matched by these options.

NOTE

Many functions in Glibc for which all aliases begin with an underscore do not follow standard function call conventions, and so should never be traced via Application Illumination.

The default is to exclude functions beginning with underscore.

(See "underscores={yes|no}" on page 5-108).

Include Functions in Standard Namespace, Exclude Functions in Standard Namespace

Includes or excludes C++ functions in the `std` namespace.

The default is to exclude C++ functions in the `std` namespace. Such functions are often inlined and so tracing them usually doesn't provide a lot of useful information.

(See "std={yes|no}" on page 5-109).

Include Functions Matching POSIX Regular Expression, Exclude Functions Matching POSIX Regular Expression

Includes or excludes functions whose names match a POSIX regular expression (see **`regex(7)`**). A function name matches the regular expression if any alias or fully qualified C++ name (if applicable) matches it. The regular expresssion must match the whole name (an implicit `^` and `$` is placed before and after the regular expression respectively).

```
<defaults>
   <option iregex=regex/>
   <option xregex=regex/>
</defaults>
```

By default

```
main,
.*\.internal_io.ada, and
.*\.internal_io\.ada\.\..*
```

are excluded.

To include only functions matching a particular *regex*, first exclude all functions:

```
--xregex=.* --iregex=regex
```

(See "iregex="regex", xregex="regex"" on page 5-109).

The context menu on an individual regular expression allows you to change their order (order is important!) or remove a regular expression from the list.

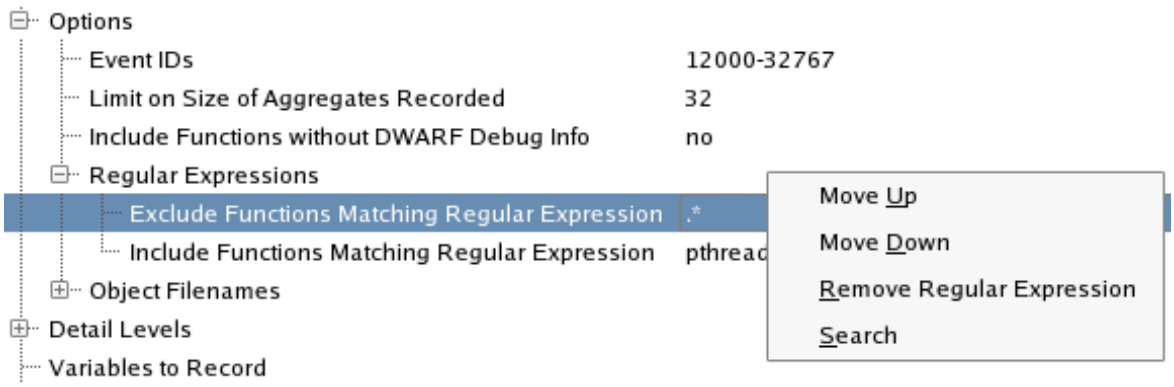


Figure 5-66. Regular Expression Context Menu

Object Filenames

Specifies *object files* that contain the functions to be illuminated. They may be a whole program, archives, shared objects, individual object files, or debug-info files. If the DWARF debug information has been placed in a separate debug-info file, it must be listed immediately after its corresponding object file.

Right click on Object Filenames to get the context menu. Browse for Object Files brings up a standard file dialog for selecting object files. Multiple object files may be selected using control and shift click.

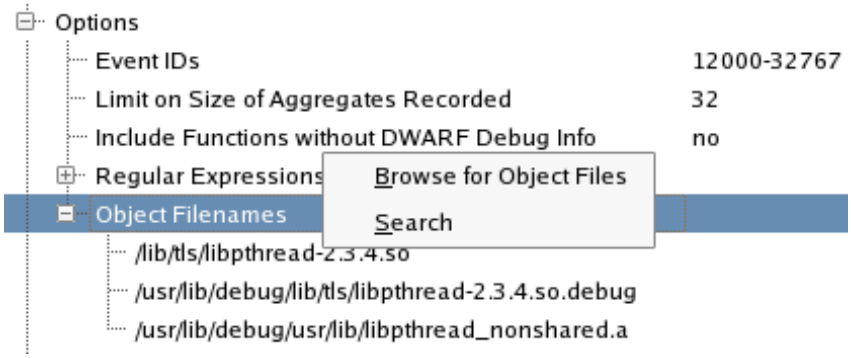


Figure 5-67. Object Filenames Context Menu

The context menu on an individual object file allows you to change their order (this is important only for debuginfo files), edit the path to the object file, or remove it from the list.

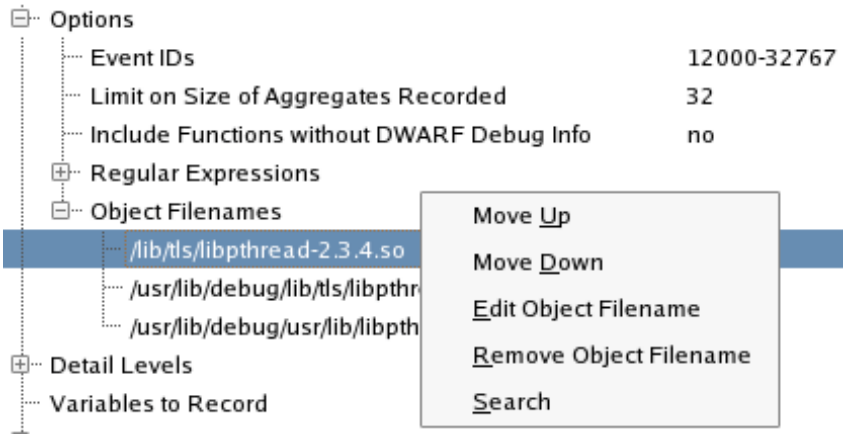


Figure 5-68. Object Filename Context Menu

The Object Filenames list is normally filled in when the session manager creates the illuminator.

(See "`filename="filename"`" on page 5-109).

Detail Levels

Named detail levels control what functions are illuminated and what details are recorded with those illuminated functions' entry and return events. By default there are three detail levels, `1`, `2`, and `3`. You may delete these and/or add more. Their names are not limited to numbers, but may be any string that can be part of a filename.

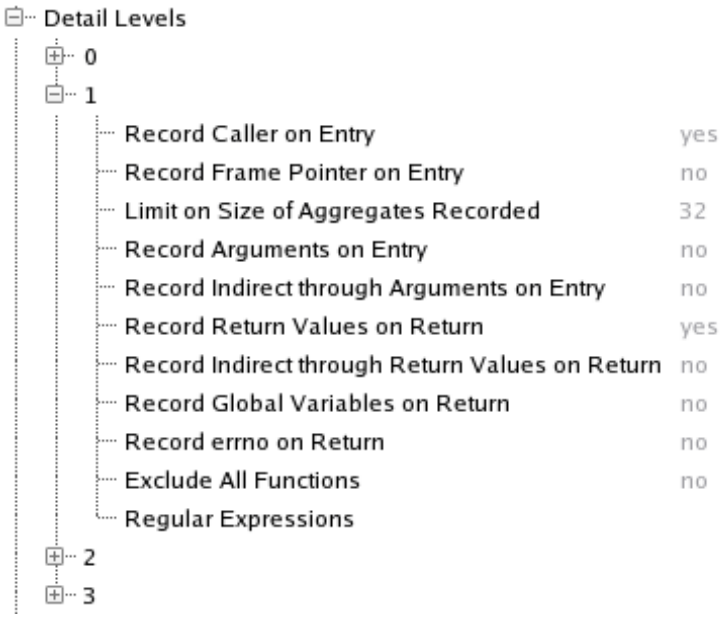


Figure 5-69. Detail Levels

There are numerous settings that can be made for each detail level. If a setting has the default value, it is displayed in gray. The default Limit on Size of Aggregates Recorded is inherited from the Limit on Size of Aggregates Recorded setting in Options. The Regular Expressions list is empty by default.

The default value for the other settings depends on the name of the detail level as detailed by the table below. To return a setting to the default setting right click on it and select Clear Setting from the context menu.

(See "level" on page 5-105).

Table 5-2. Detail Levels Settings Defaults

Attribute	`1`	`2`	`3`	Custom
Record Caller on Entry	`yes`	`yes`	`yes`	`no`
Record Frame Pointer on Entry	`no`	`yes`	`yes`	`no`
Record Arguments on Entry	`no`	`yes`	`yes`	`no`
Record Indirect through Arguments on Entry	`no`	`no`	`yes`	`no`

Table 5-2. Detail Levels Settings Defaults

Attribute	1	2	3	Custom
Record Return Values on Return	yes	yes	yes	no
Record Indirect through Return Values on Return	no	no	yes	no
Record Global Variables on Return	no	no	yes	no
Record errno on Return	no	no	yes	no
Exclude All Functions	no	no	no	no

Record Caller on Entry

Records the return address on entry events. (See "caller={yes|no}" on page 5-106).

Record Frame Pointer on Entry

Records the callers's frame pointer on entry events. (See "frame={yes|no}" on page 5-106).

Limit on Size of Aggregates Recorded

Sets a size limit (in bytes) on aggregate values recorded with entry and return events. The aggregate value recorded is truncated beyond the limit. (See "aggregate_limit=limit" on page 5-106).

Record Arguments on Entry

Records a function's arguments on entry events. If the argument is an aggregate type (class, structure, union, or array), only a limited number of bytes will be recorded. This limit is set by the Limit on Size of Aggregates Recorded setting. (See "args={yes|no}" on page 5-106).

Record Indirect through Arguments on Entry

Records the value pointed to by a function's pointer arguments on entry events. If the argument is a pointer to an aggregate type (class, structure, union, or array), only a limited number of bytes will be recorded. This limit is set by the Limit on Size of Aggregates Recorded setting. (See "addr_args={yes|no}" on page 5-107).

Record Return Values on Return

Records a function's return value and out arguments on return events. If the value is an aggregate type (class, structure, union, or array), only a limited number of bytes will be recorded. This limit is set by the Limit on Size of Aggregates Recorded setting. (See "return_val={yes|no}" on page 5-107).

Record Indirect through Return Values on Return

Records the value pointed to by a function's pointer return value and pointer out arguments on return events. If the value is an aggregate type (class, structure, union, or array), only a limited number of bytes will be recorded. This limit is set by the Limit on Size of Aggregates Recorded setting. (See "addr_ret={yes|no}" on page 5-107).

Record Global Variables on Return

Records select global variables and indirection through select global variables on return events. If the value is an aggregate type (class, structure, union, or array), only a limited number of bytes will be recorded. This limit is set by the Limit on Size of Aggregates Recorded setting. The list of variables is empty by default. See "Variables to Record" on page 5-87, "Select Variables to Record, Add Variable to Record" on page 5-93, "Select Variables to Record Add Variable to Record" on page 5-97, "variables={yes|no}" on page 5-107).

Record errno on Return

Records the value of errno on return events. (See "errno={yes|no}" on page 5-107).

Exclude All Functions

Excludes all functions from having entry and return events recorded for them at this detail level. This can be convenient for restricting a detail level to a small set of functions by then overriding this setting for individual groups or functions. (See "exclude={yes|no}" on page 5-107).

Regular Expressions

Excludes or includes select functions from having entry and return events recorded from them. The same regular expressions may be used here as in the Options section (see "Regular Expressions" on page 5-81). Functions cannot be included here that were excluded in the Options sections. The inclusion regular expressions are for putting back functions that were excluded by previous regular expressions. For example, you could exclude ".*", then include "c_.*" to restrict this detail level to just those functions starting with "c_". But if the Options section had excluded functions matching "c_a.*", they would not be reincluded. (See "underscores={yes|no}" on page 5-108, "std={yes|no}" on page 5-109, "iregex="regex", xregex="regex"" on page 5-109).

Variables to Record

Detail level 3 (by default) and any detail level with the Record Global Variables on Return setting turned on will record any global variables or indirection through global variables that are listed in this section on function return events. The function must have the global variable declared in its DWARF debugging information. No error is generated

for functions that don't have the global variable in their DWARF, they just don't record the variable in their return events.

Right click on Variables to Record and select Add Variable or Select Variables to Record to add variables or indirection through variables to this list. (See "variable" on page 5-109).

Figure 5-70. Variables to Record

Add Variable

Brings up a dialog to allow you to type in a variable name. If the variable name is preceded by an asterisk ("*"), then the value pointed to by the variable, if it is a pointer, is recorded instead. If the variable isn't a pointer and indirection is requested, no value is recorded.

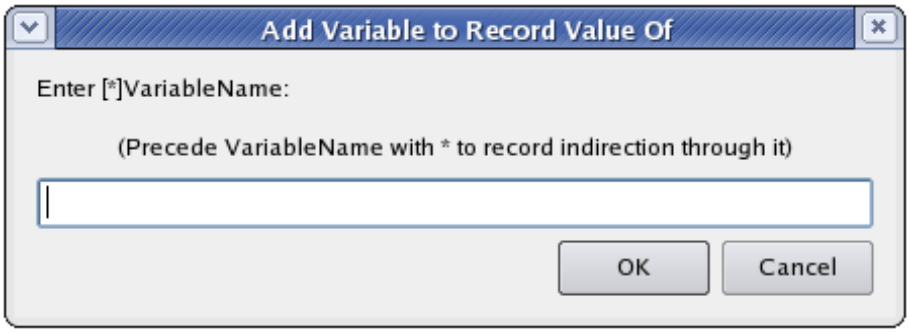


Figure 5-71. Add Variable Dialog

Select Variables to Record

Brings up a dialog with a list of variables and their types that were discovered by populating the illuminator (see "Populate" on page 5-45). Pointer variables will be in the list twice: once for themselves and once for indirection through them. Select or deselect the variables desired by clicking in the check box next to them.

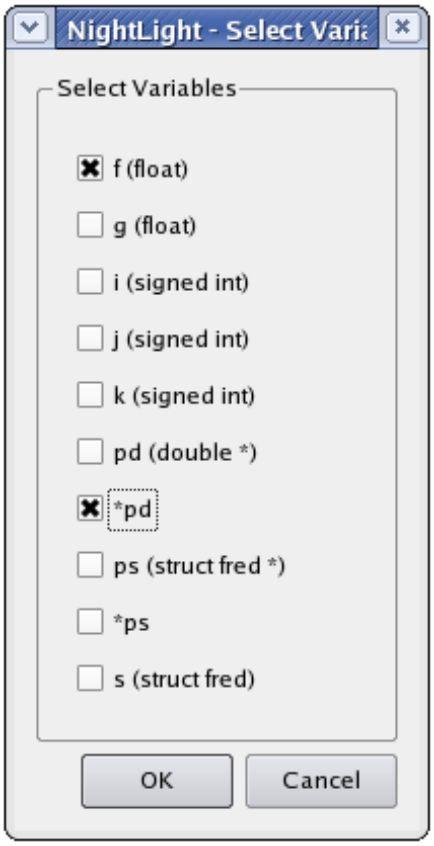


Figure 5-72. Select Variables to Record Dialog

To remove a variable from this list, use the Select Variables to Record dialog to deselect them, or right click on the variable to be removed and select the Remove Variable context menu item. This context menu may also be used to rearrange the variables in the list.

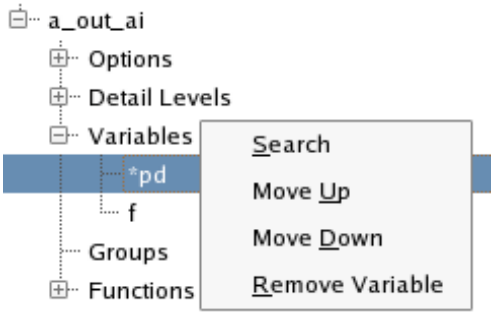


Figure 5-73. Variable Name Context Menu

Groups

Functions may be placed in named groups. This is convenient for applying customizations. Perhaps, for example, you want more details on the functions in the group `iconv`. You could create a copy of detail level `1` called `iconv_details`, and then customize that detail level to include more details for functions in the `iconv` group. (See "group" on page 5-104).

Create a Group

To create a group, right click on Groups and select New Group from the context menu that pops up. A dialog will pop up asking for a name for the group.

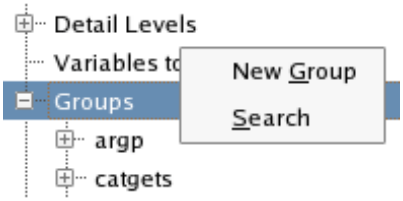


Figure 5-74. Groups Context Menu

Customize a Group

Right click on the group name and select an item from the context menu that pops up.

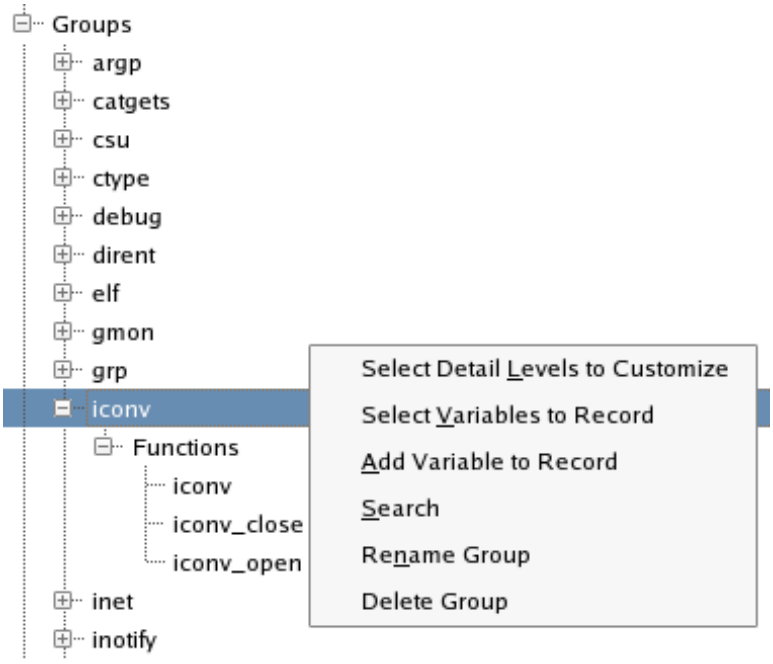


Figure 5-75. Group Name Context Menu

Select Detail Levels to Customize

This allows you to customize a detail level for a particular group. A dialog pops up allowing you to select or deselect which detail levels you want to customize.

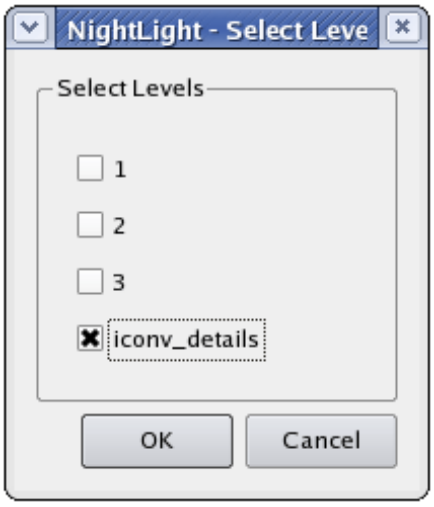


Figure 5-76. Select Detail Level to Customize Dialog

An additional branch is added to the group's tree for each customized detail level. The values for each detail level setting are gray when they are inherited from the Detail Levels section. The Exclude This Group setting overrides the Exclude All Functions setting in the Detail Levels section. To create a custom detail level that only records events for one group's functions, set Exclude All Functions in the Detail Levels section to `yes`, then override that for a particular group by setting Exclude This Group to `no`.

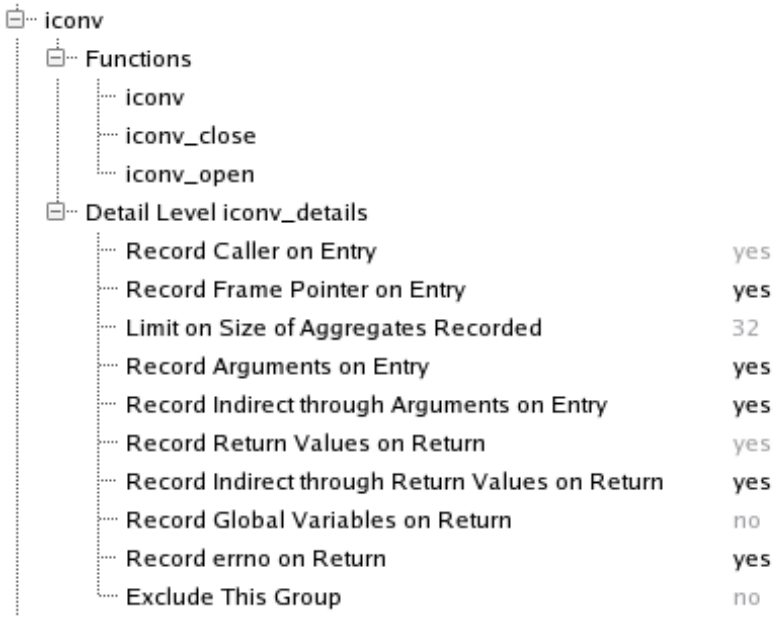


Figure 5-77. A Customized Custom Detail Level for a Group

Select Variables to Record, Add Variable to Record

These allow you to record additional global variables for return events of just this group's functions for detail levels that have Record Global Variables on Return `true`. The same dialogs are brought up to select variables as in the Variables to Record section (see "Variables to Record" on page 5-87).

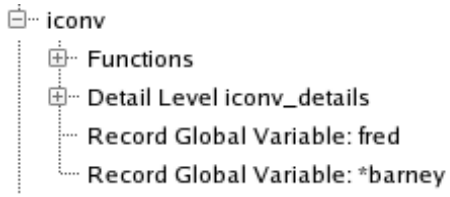


Figure 5-78. A Group with Additional Variables to Record

Rename Group

This pops up a dialog that prompts for a new name for the group.

Delete Group

This deletes a group.

Selecting Members of a Group

To select which functions are in a group, right click on the Functions branch under a group name and select the Select Functions in Group context menu item.

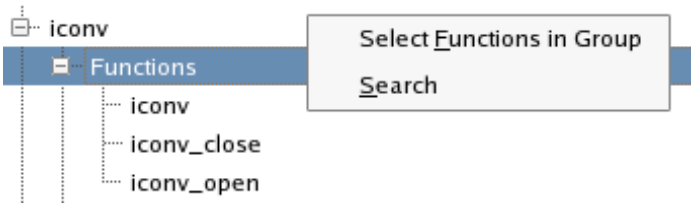


Figure 5-79. Member Functions Context Menu

This brings up a dialog with a list of all functions defined in the Functions section to select or deselect from.

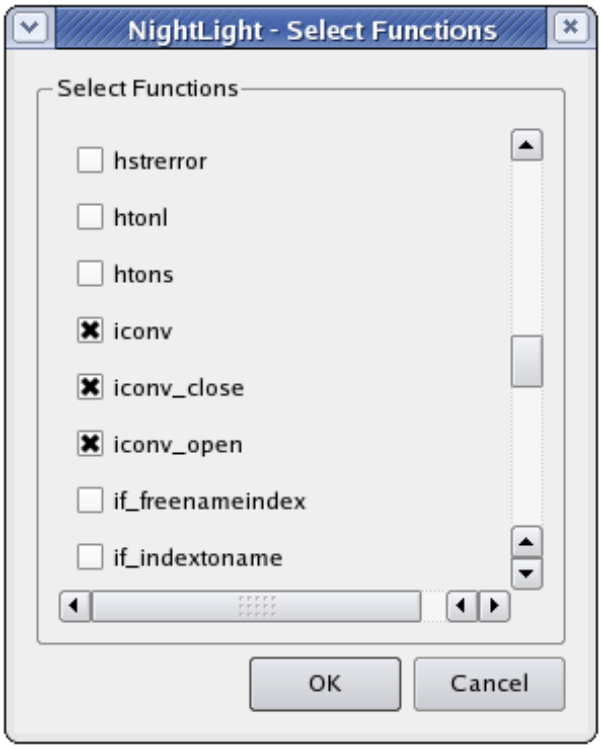


Figure 5-80. Select Functions Dialog

Functions may also be added to a group from the Functions section (see "Adding a Function to a Group" on page 5-99).

Functions

The Function section allows customization of individual functions. A function does not have to be listed here to be illuminated (although it does need to be here to be a member of a group). (See “function” on page 5-103).

Adding a Function

Functions are usually added to this section by using the populate command in the session manager (see “Populate” on page 5-45) or **`nlight --populate`** on a command line (see “nlight --populate” on page 5-71). Functions may also be added by right clicking on Functions and selecting Add a Function from the context menu that pops up. A dialog will pop up asking for the function name.

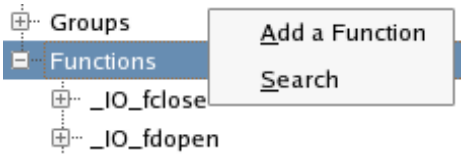


Figure 5-81. Functions Context Menu

Customizing a Function

To customize a function, right click on the function name and choose an item from the context menu that pops up.

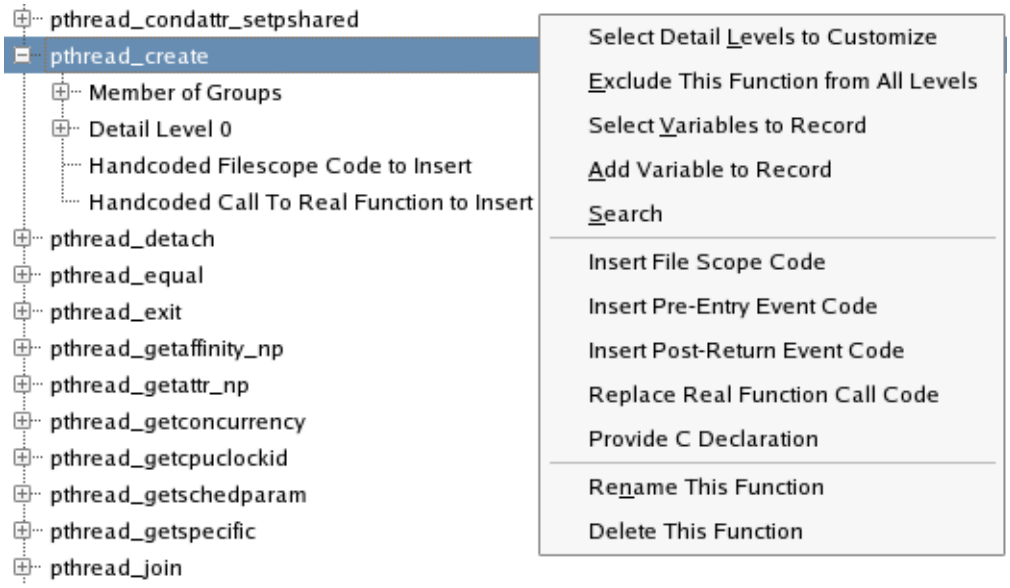


Figure 5-82. Function Context Menu

Select Detail Levels to Customize

This allows you to customize a detail level for a particular function. It works just like customizing a detail level for a particular group (see "Select Detail Levels to Customize" on page 5-92).

Exclude This Function from All Levels

This allows you to prevent this function from being illuminated for all detail levels. Another way to do this would be to use a regular expression to exclude the function in the Options section.

To remove the exclusion, right click on Exclude This Function From All Detail Levels and select Remove Exclusion from the context menu that pops up.

(See "exclude" on page 5-103).

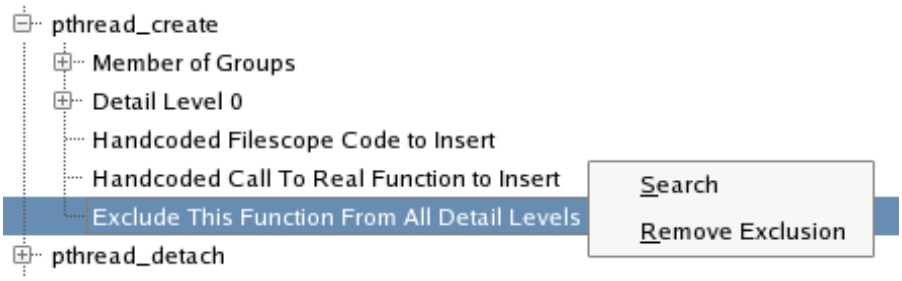


Figure 5-83. Remove Exclusion Context Menu Item

Select Variables to Record
Add Variable to Record

These allow you to record additional global variables for just the return event of this function for detail levels that have Record Global Variables on Return `true`. The same dialogs are brought up to select variables as in the Variables to Record section (see “Variables to Record” on page 5-87). Settings in gray are inherited from the Groups section (see “Select Variables to Record, Add Variable to Record” on page 5-93). If a function is a member of more than one group, the first group in the list that provides an explicit setting is the effective value.

Insert File Scope Code
Insert Pre-Entry Event Code
Insert Post-Return Event Code
Replace Real Function Call Code

These are advanced items for inserting assembly code fragments in the function “wrapper” code that records the entry and return events. To edit the code that is to be inserted, double click on the appropriate Handcoded item or right click on it and select Edit from the context menu that pops up. This brings up a text editor dialog. See “wrapper_file_scope” on page 5-110, “wrapper_post” on page 5-110, “wrapper_pre” on page 5-110, “wrapper_real” on page 5-111 for more detailed documentation on these code fragments.

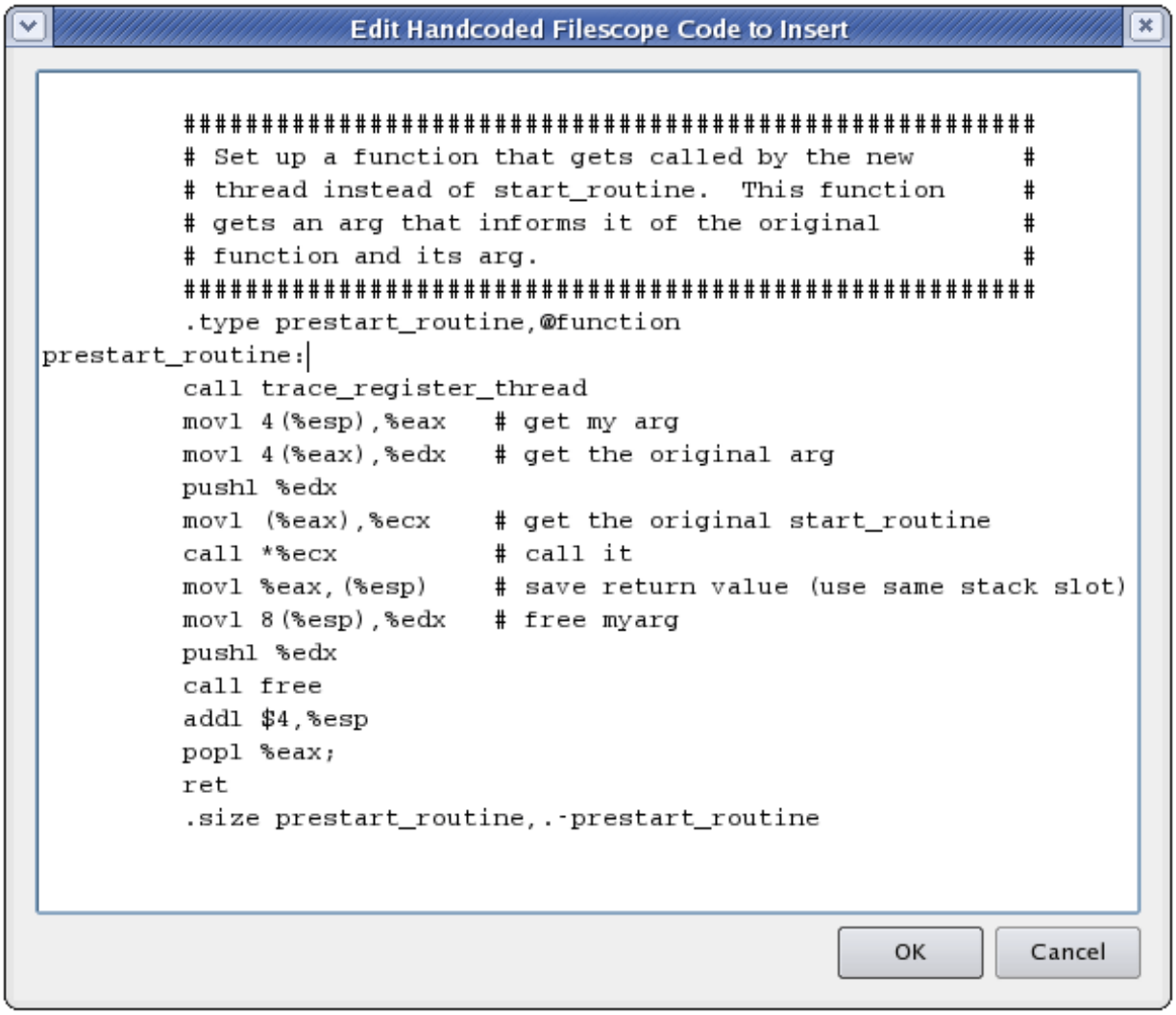


Figure 5-84. Edit Handcoded Dialog

Provide C Declaration

Provides a C language declaration for functions that do not have DWARF debug information (perhaps the function was written in assembly, for example). This setting is ignored if the function has DWARF debug information. The declaration may be preceded by `#includes` and type definitions. The declaration itself should not include an `extern`, nor be terminated by a semi-colon. (See "declare" on page 5-102).

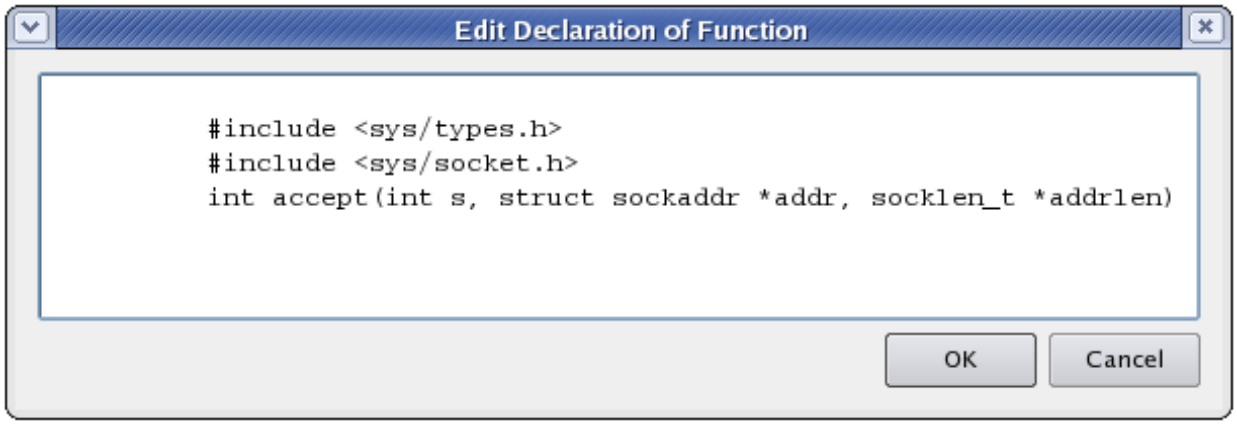


Figure 5-85. Edit Declaration Dialog

Rename This Function

Pops up a dialog that prompts for a new name for the function.

Delete This Function

Deletes a function from the Functions section. This does not stop the function from being illuminated, it only removes the customizations for the function and the function's group memberships.

Adding a Function to a Group

To add a function to a named group of functions that is defined in the Groups section, right click on the Groups branch under a function name and select the Select Groups context menu item.

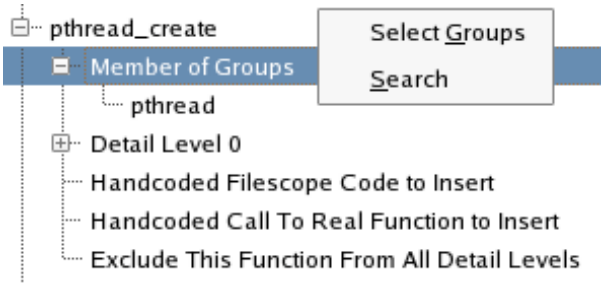


Figure 5-86. Member Groups Context Menu

This brings up a dialog with a list of all groups defined in the Groups section to select or deselect from.

Figure 5-87. Select Groups Dialog

Functions may also be added to a group from the Groups section (see "Selecting Members of a Group" on page 5-94).

Customizing an Illuminator by Editing the config.xml File

The **`config.xml`** file in the illuminator directory may be edited to customize the illuminator. This section provides a brief dictionary for the supported XML elements. Each element is documented in alphabetical order and is headed with a brief synopsis that shows the context in which it appears, as well as other elements in may contain.

comments

Comments (<!-- comment -->) may be placed amongst the XML using standard XML comment syntax. Elements that enclose text (such as `<declare>`, `<wrapper>` and `<wrapper_*>` may not have comments embedded in the text. Comments are lost when a **`config.xml`** file is repopulated with the **`nlight --populate`** command. There is no guarantee on the order of the elements, so there is no way to know exactly where to place the comments in the repopulated file. The three-way diff tool, **`diff3(1)`**, may be used to help reinsert them into the approximate correct place.

config

```
<config>
   [<defaults>
      [<level .../> ...]
      [<options .../> ...]
      [<variable name=[*]variable_name/> ...]
   </defaults> ...]
   [<variable name=variable_name
               [type=type_name ptr={yes|no}]/> ...]
   [<group name=group_name>
      [<variable name=[*]variable_name/> ...]
   </group> ...]
   [<function name=function_name>
      [<exclude/>]
      [<level ... /> ...]
      [<group name=group_name/> ...]
      [<wrapper>wrapper function</wrapper>]
      [<wrapper_file_scope>some code</wrapper_file_scope>]
      [<wrapper_pre>some code</wrapper_pre>]
      [<wrapper_real>call to real function</wrapper_real>]
      [<wrapper_post>some code</wrapper_post>]
      [<declare>declaration</declare>]
      [<variable name=[*]variable_name/> ...]
   </function> ...]
</config>
```

Encloses the entire file. It may contain four types of elements: `<defaults>` (see page 5-102), `<variable>` (see page 5-109), `<group>` (see page 5-104), and `<function>` (see page 5-103).

declare

```
<function ...>
   <declare>declaration</declare>
</function>
```

Provides a C language declaration for functions (see "function" on page 5-103) that do not have DWARF debug information (perhaps the function was written in assembly, for example). This element is ignored if the function has DWARF debug information. The declaration may be preceded by `#include`s and type definitions. The declaration itself should not include an `extern`, nor be terminated by a semi-colon. Here is an example:

```
<declare>
   #include &lt;sys/types.h&gt;
   pid_t getpgid(pid_t pid)
</declare
```

Certain characters are special in XML and must be replaced with "character entities":

Table 5-3. Character Entities

`&`	`&`
`<`	`<`
`>`	`>`
`&quot;`	`"`
`&apos;`	`'`

defaults

```
<config>
   <defaults>
      [<level name=level_name
             [caller={yes|no}]
             [frame={yes|no}]
             [aggregate_limit=limit]
             [args={yes|no}]
             [addr_args={yes|no}]
             [return_val={yes|no}]
             [addr_ret={yes|no}]
             [variables={yes|no}]
             [errno={yes|no}]
             [exclude={yes|no}]>
      [<options [underscores={yes|no}]
                [std={yes|no}]
                [xregex=regex]
                [iregex=regex]/> ...]
      </level> ...]
      [<options .../> ...]
```

```
      [<variable name=[*]variable_name/> ...]
   </defaults>
</config>
```

Defines the defaults for all functions and groups (see "config" on page 5-101). It may contain zero or more `<level>` elements (see "level" on page 5-105) to customize the detail levels `1`, `2`, or `3`, or to define a user-named custom detail level. It may contain zero or more `<options>` elements (see "options" on page 5-108) to specify values for certain command line options.

Finally, it may contain zero or more `<variable>` elements (see "variable" on page 5-109) to specify global variables to be recorded with the return event for any function whose DWARF defines the global variables when the detail level includes variables.

exclude

```
<function ...>
   <exclude/>
</function>
```

Excludes a function (see "function" on page 5-103) from all detail levels without having to list separate `<level>` (see "level" on page 5-105) elements. If both the `<exclude/>` element and an `exclude` attribute (see "`exclude={yes|no}`" on page 5-107) for a specific `<level>` are specified in a `<function>` element, the exclude attribute takes precedence. Thus:

```
    <function name=hello>
       <exclude/>
       <level=3 exclude=no>
    </function>
```

will exclude `hello()` from all detail levels except `3`.

function

```
<config>
   <function name=function_name>
      [<exclude/>]
      [<level ... /> ...]
      [<group name=group_name/> ...]
      [<wrapper>wrapper function</wrapper>]
      [<wrapper_file_scope>some code</wrapper_file_scope>]
      [<wrapper_pre>some code</wrapper_pre>]
      [<wrapper_real>call to real function</wrapper_real>]
      [<wrapper_post>some code</wrapper_post>]
      [<declare>declaration</declare>]
      [<variable name=[*]variable_name/> ...]
   </function>
</config>
```

Defines settings for a specific function (see "config" on page 5-101). It may contain:

- zero or more <level> elements (see "level" on page 5-105) to override the defaults for the detail levels for *function_name*;
- zero or more <group> elements (see "group" on page 5-104) to designate *function_name* as a member of a group of functions;
- an optional <wrapper> element (see "wrapper" on page 5-110) to provide a hand written "wrapper" function;
- optional <wrapper_*> elements (see "wrapper_file_scope" on page 5-110, "wrapper_post" on page 5-110, "wrapper_pre" on page 5-110, and "wrapper_real" on page 5-111) to provide some code to insert into or replace parts of the machine generated "wrapper" function;
- an optional <declare> element (see "declare" on page 5-102) to provide the declaration of the function being "wrapped";
- zero more more <variable> elements (see "variable" on page 5-109) to specify global variables to be recorded with return events if the function's DWARF defines the global variables when the detail level includes variables.

group

```
<config>
   <group name=group_name>
      [<level ... /> ...]
      [<variable name=[*]variable_name/> ...]
   </group>
</config>
```

Defines settings for a named group of functions (see "config" on page 5-101). It may contain zero or more <level> elements (see "level" on page 5-105) to specify settings for particular detail levels for the named group of functions. The named levels must be one of the three predefined levels, or a user-named custom level defined in a defaults element.

It may also contain zero or more more <variable> elements (see "variable" on page 5-109) to specify global variables to be recorded with return events for all functions in the group whose DWARF defines the global variables when the detail level includes variables.

```
<function ...>
   <group name=group_name/>
</function>
```

Designates in a <function> element (see "function" on page 5-103) that the subject function is a member of *group_name*. In this context it may not contain any <level> or <variable> elements.

level

```
<defaults>
   <level name=level_name
          [caller={yes|no}]
          [frame={yes|no}]
          [aggregate_limit=limit]
          [args={yes|no}]
           [addr_args={yes|no}]
           [return_val={yes|no}]
          [addr_ret={yes|no}]
          [variables={yes|no}]
          [errno={yes|no}]
          [exclude={yes|no}]>
      [<options [underscores={yes|no}]
                [std={yes|no}]
                [xregex=regex]
                [iregex=regex]/>]
   </level>
</defaults>
```

Modifies the default settings (see “defaults” on page 5-102) for predefined detail levels or defines a custom detail level. The attributes and elements control whether a function is traced, and what details are recorded with the trace events if it is.

<options> elements (see “options” on page 5-108) corresponding to **--x*** and **--i*** command line options may also be specified in a <level> element when it appears in a <defaults> element. These may not be used to include any functions that were excluded at the command line level or by the corresponding <options> element within a <defaults> element, but may be used to restrict a level to a smaller subset for a specific detail level. One way of creating a new level that exludes all functions but one is:

```
<defaults>
   <level name=0>
      <options xregex=".*" iregex="pthread_create"/>
   </level>
</defaults>
```

The effective value of each attribute for a given function and detail level is determined by searching for a definition of the attribute in the following places in the following order:

- a <level> element in the function’s <function> element;
- a <level> element in each of the function’s group memberships, in the order the <group> elements were listed;
- a <level> element in the <defaults> element;
- the system defaults.

The system defaults for the attributes are:

Table 5-4. System Defaults

Attribute	Level 1	Level 2	Level 3	Custom Levels
caller	yes	yes	yes	no
frame	no	yes	yes	no
aggregate_limit	16	16	16	16
args	no	yes	yes	no
addr_args	no	no	yes	no
return_val	yes	yes	yes	no
addr_ret	no	no	yes	no
variables	no	no	yes	no
errno	no	no	yes	no
exclude	no	no	no	no

The details that can be recorded are partitioned into several named classes. To turn on one of those classes, specify *classname*=yes as an attribute to the <level> element. For example, to create a custom detail level to record only the function arguments, you would code the following element in a <defaults> element:

```
<level name="argsonly" args=yes/>
```

To turn off an attribute specify *attribute*=no.

caller={yes|no}

The return address in the caller is recorded on entry events.

frame={yes|no}

The address of the frame of the caller is recorded on entry events.

aggregate_limit=*limit*

A limit is set on the number of bytes of an aggregate that can be recorded with an entry or return event. The limit must be at least 16 bytes.

args={yes|no}

The arguments passed to the traced function are recorded on entry events, and out arguments are recorded on return events..

`addr_args={yes|no}`

The variables pointed to by arguments that are pointers are recorded on entry events. The variables pointed to by out arguments that are pointers are recorded on return events. When these are aggregates (strings, arrays, structures, or unions), the number of bytes that may be recorded is limited by the `aggregate_limit` setting.

`return_val={yes|no}`

The return value of the function (if it has one) is recorded on return events.

`addr_ret={yes|no}`

The variable pointed to by the return value, if it is a pointer, is recorded on return events. When this is an aggregate (string, array, structure, or union), the number of bytes that may be recorded is limited by the `aggregate_limit` setting.

`variables={yes|no}`

Variables or indirection through variables specified with `<variable>` elements (see "variable" on page 5-109) in `<defaults>`, `<group>`, and `<function>` elements are recorded on return events.

`errno={yes|no}`

The value of `errno` is recorded on return events.

`exclude={yes|no}`

Functions are entirely excluded from being recorded. Normally this would be set to yes only on individual functions or groups of functions. Or, one could set it to `yes` in `<defaults>`, then override that on individual functions or groups of functions in order to only include those functions. For example, the following creates a new detail level that excludes all but one function:

```
<defaults>
   <level name=0 exclude=yes/>
</defaults>
<function name=pthread_create>
   <level name=0 exclude=no/>
</function>
```

See also "exclude" on page 5-103 for a shorthand way to exclude a function from all detail levels.

options

```
<defaults>
   <options [event_ids="N-[M]"]
            [aggregate_limit="limit"]
            [nodebug={yes|no}]
            [underscores={yes|no}]
            [std={yes|no}]
            [xregex="regex"]
            [iregex="regex"]
            [filename="filename"]
   />
</defaults>
```

Specifies values for several command line options (see "defaults" on page 5-102, "nlight --create" on page 5-68). Options specified after a **`--config`** option on the command line will override those set in the *config.xml* file.

```
<defaults>
   <level name=level_name ...>
     [<options [underscores={yes|no}]
               [std={yes|no}]
               [xregex=regex]
               [iregex=regex]/>]
   </level>
</defaults>
```

Specifies level-specific overrides for command line options that exclude or include functions by their name (see "level" on page 5-105, "`--i*, --x*`" on page 5-70).

`event_ids="`*N*`-[`*M*`]"`

Specifies the range of event_ids to be mapped to entry and return events (see "`--event_ids=`N-[M]" on page 5-69).

`aggregate_limit="`*limit*`"`

Limits the number of bytes of an aggregate that may be recorded with an event (see "`--aggregate_limit=`limit" on page 5-68). The limit must be at least 16 bytes.

`nodebug={yes|no}`

Specifies whether function names that have no debug information are to be included or excluded respectively (see "`--do_nodebug, --dont_nodebug`" on page 5-69).

`underscores={yes|no}`

Specifies whether function names that start with an underscore are to be included or excluded respectively (see "`--iunderscores, --xunderscores`" on page 5-70). This may also be specified for a particular level (see "level" on page 5-105).

`std=`*`{`*`yes|no`*`}`*

Specifies whether function names in the C++ `std` namespace are to be included or excluded respectively (see "`--istd, --xstd`" on page 5-71). This may also be specified for a particular level (see "level" on page 5-105).

`iregex="`*`regex`*`", xregex="`*`regex`*`"`

Specifies whether function names that match the POSIX regular expression are to be included or excluded respectively (see "`--iregex=`regex`, --xregex=`regex" on page 5-70). This may also be specified for a particular level (see "level" on page 5-105).

To specify multiple instances of these attributes, you must use separate `<options>` elements since XML syntax does not allow duplicate attribute names.

`filename="`*`filename`*`"`

Specifies an object file, shared object file, debug-info file, archive, or program to read DWARF from to generate "wrapper" functions. These filenames may also be specified as arguments to the **`nlight --create`** command (see "nlight --create" on page 5-68).

To specify more than one filename, you must use multiple `<options>` elements since XML syntax does not allow duplicate attribute names.

variable

```
<config>
   <variable name=variable_name [type=type_name ptr={yes|no}]/>
</config>
```

Defines a a global variable (see "config" on page 5-101). **`illuminator`** does not actually use this element. It is populated by the **`nlight --populate`** command (see "nlight --populate" on page 5-71). You may wish to consult this list (or **`nlight --report`** output, see "nlight --report" on page 5-72) to get the exact correct spelling of certain variable names in name-mangling languages. The fully qualified name is reconstructed from the mangled name, and may include elements that are implicit in the original source.

```
<{defaults|group|function}>
   <variable name=[*]variable_name/>
</{defaults|group|function}>
```

Names a variable (with optional indirection), when it appears in a `<defaults>`, `<group>`, or `<function>` element (see "defaults" on page 5-102, "group" on page 5-104, "function" on page 5-103), that will be recorded on return events at detail levels that have the `variables=yes` attribute set (see "`variables={yes|no}`" on page 5-107). Depending on which element it appears in, it may apply to all functions, all functions in a group, or a particular function (for `<defaults>`, `<group>`, or `<function>` elements respectively). The function's DWARF must include a definition of the variable in question. No error message is generated if it is absent from the DWARF.

wrapper

```
<function ...>
   <wrapper>assembly "wrapper" function</wrapper>
</function>
```

Specifies a hand coded "wrapper" function for a specific function (see "function" on page 5-103). The text between the opening and closing tags is copied verbatim into the "wrapper" function assembly language source file. It may not be used with the other <wrapper_*> elements.

wrapper_file_scope

```
<function ...>
   <wrapper_file_scope>some code</wrapper_file_scope>
</function>
```

Specifies assembly language code to be inserted in "file scope" just before the "wrapper" function (see "function" on page 5-103). It may not be used with a <wrapper> element.

wrapper_post

```
<function ...>
   <wrapper_post>some assembly code</wrapper_post>
</function>
```

Specifies assembly language code to insert into a generated "wrapper" function after the return event is recorded but just before actually returning (see "function" on page 5-103). One use might be to insert some debug code into the application. It may not be used with a <wrapper> element.

wrapper_pre

```
<function ...>
   <wrapper_pre>some assembly code</wrapper_pre>
</function>
```

Specifies assembly language code to insert into a generated "wrapper" function before the entry event is recorded (see "function" on page 5-103). One use might be to test for a situation where you don't want an event to be recorded. It may not be used with a <wrapper> element.

wrapper_real

```
<function ...>
   <wrapper_real>assembly code call to real function</wrapper_real>
</function>
```

Specifies assembly language code to call the real function in place of the default code in a generated "wrapper" function (see "function" on page 5-103). It may not be used with a `<wrapper>` element.

Here's an example of intercepting a function called through a pointer parameter in `pthread_create()` in order to call `trace_register_thread()` in the newly created thread:

```
<function name=pthread_create>
   <wrapper_file_scope>
         ########################################################
         # Set up a function that gets called by the new       #
         # thread instead of start_routine.  This function     #
         # gets an arg that informs it of the original         #
         # function and its arg.                               #
         ########################################################
         .type prestart_routine,@function
prestart_routine:
         pushq %rdi;         # save the arg while I do a call
         call trace_register_thread
         movq (%rsp),%rax    # get the arg back
         movq 8(%rax),%rdi   # get the original arg
         movq (%rax),%r11    # get the original start_routine
         call *%r11          # call it
         pushq %rax          # save return value
         movq 8(%rsp),%rdi   # free myarg
         call free
         popq %rax;
         addq $8,%rsp
         ret
         .size prestart_routine,.-prestart_routine
   </wrapper_file_scope>
   <wrapper_real>
         # allocate arg for the interceptor routine (thread safe)
         movq $16,%rdi
         call malloc

         # store the  original start_routine
         # and arg into the new arg
         movq -24(%rbp),%r11               # start_routine
         movq %r11,(%rax)
         movq -32(%rbp),%r11               # arg
         movq %r11,8(%rax)

         # set up parameters to the interceptor routine
         movq -8(%rbp),%rdi                # newthread
         movq -16(%rbp),%rsi               # attr
         lea  prestart_routine(%rip),%rdx  # interceptor start
                                           #    routine
         movq %rax, %rcx                   # myarg

         # call the real function passing my interceptor routine
         call __real_pthread_create
   </wrapper_real>
</function>
```

Note that to call the real function from a "wrapper" you call `__real_function`, otherwise, the call to function would be diverted to `__wrap_function` and become an infinite recursion.

The NightTrace function `trace_register_thread()` is obsolete in the latest NightTrace release, but is retained in this example because it makes such a good illustration of doing something complex with inserting code in an illuminator.

6
Performance Tuning

The NightTrace default configuration is often sufficient for most tracing needs, however, situations with exceptionally high trace event rates or those requiring precise control over disk activity may require adjustment. This chapter discusses the following:

- "Preventing Trace Event Loss" on page 6-1
- "Conserving Disk Space" on page 6-3
- "Conserving Memory and Accelerating ntrace" on page 6-3

Preventing Trace Event 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 as long as it can copy the shared memory buffers to the output device faster than the application or kernel can fill up all remaining shared memory buffers.

NightTrace reports lost trace events in several ways:

- The **`--info`** options to **`ntraceud`** and **`ntracekd`** describe the number of lost events
- The Daemon Control area in **`ntrace`** displays event loss counts
- NightTrace display pages include a visual indicator on the ruler, a capital L character, indicating where event loss started to occur
- An internal trace point, `NT_LOST_DATA`, is included in the trace data output at the point where trace events began to be lost

NOTE

Events that are overwritten in file-wrap and buffer-wrap modes are not considered lost events and are not reported.

Daemon Scheduling Adjustment

The scheduling policy, priority, and CPU bias of daemons can be adjusted using the following methods:

- Invoke **ntraceud** and **ntracekd** with the **--priority=***P*, **--policy=***P*, and **--processor=***C* command line options to select scheduling priority, policy and CPU binding.
- Select the scheduling policy, scheduling priority and CPU bias from the Runtime tab of the Daemon Definition dialog in the **ntrace** tool.

Increasing Trace Buffer Size

The number of trace buffers and the size of trace buffers can be adjusted using the following methods:

- Specify larger values using the **--numbufs** and **--buflen** options to **ntraceud**. The default values for these options are 8 and 32768, respectively.
- Specify larger values for the *ntc_num_buffers* and *ntc_buffer_length* fields in the `ntconfig_t` configuration record passed to `trace_begin`. The default values for these fields are 8 and 32768, respectively. Note that these configuration values will be ignored if the corresponding user daemon has already started and the value of *ntc_daemon_preferred* is set to `TRUE`.
- Specify larger values using the **--numbufs** and **--bufsize** options to **ntracekd**. The default values for these options are 4 and 50000, respectively.
- Specify larger values for Number of Buffers and Buffer Size in the User Trace tab of the Daemon Definition dialog in the **ntrace** tool. The default values for these settings are 8 and 32768, respectively.
- Specify larger values for Number of Trace Buffers and Trace Buffer Size using the Other tab of the Daemon Definition dialog in the **ntrace** tool. The default values for these settings are 4 and 50000, respectively.

When increasing user trace buffer sizes, your request may be rejected if the total trace buffer shared memory size exceeds system limitations. You can increase the system shared memory limits by adjusting the *kernel.shmmax* and *kernel.shmall* variables using the **systctl(8)** command.

For user trace buffers, the number of buffers and buffer length must be individually a power of two. These values are automatically increased to the next highest power of two if this is not the case.

Since daemons are notified immediately when a single trace buffer fills, adding additional buffers is sometimes as effective as increasing the size of buffers. The kernel and applications continue to log trace events to the next shared memory buffer while the daemon flushes the filled buffer.

Programmatic Flushing

For applications which log trace events, the `trace_flush` API routine can be used to cause the associated user daemon to wake up and flush all filled buffers.

Modifying the sizes and number of trace buffers as described in the previous section is usually more effective than relying on `trace_flush`, since the daemon automatically wakes and empties buffers as individual buffers are filled.

Conserving Disk Space

If disk space is an important consideration and you are most interested in the latest events that are logged, use of file-wrap and buffer-wrap modes is helpful.

In buffer-wrap mode, no disk activity occurs until the daemon is terminated or an explicit flush is requested. When all trace buffers are filled, the oldest events are overwritten by the newest events.

In file-wrap mode, a file size maximum is imposed and the oldest events are overwritten by the newest events when the maximum size is reached.

Both of these options can be useful when desiring to obtain trace data from a situation which rarely appears.

For example, the following commands might be used to capture kernel and user trace data for an extended period of time (even hours or days) until your application detects a specific situation:

```
> ntracekd --size=20M kernel-data
> ntraceud --filewrap=10M user-data
> ./a.out
> ntraceud --quit user-data
> ntracekd --quit kernel-data
```

When capturing kernel data from the ntrace graphical analysis tool and streaming the data for immediate analysis, buffer-wrap mode is also very useful.

The Linux kernel can generate huge numbers of events on busy systems. Use of buffer-wrap mode allows you to take snapshots of kernel data for immediate analysis or to be saved for future analysis. Select the Buffer Wrap option on the General tab of the Daemon Definition dialog and subsequently press the Flush button in the Daemon Control area of the NightTrace Main window when you wish to sample kernel data.

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 a 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 be slow. To conserve memory and accelerate **ntrace**:

- Log only trace events you are really interested in.
- Disable uninteresting events via the **--disable** option to **ntraceud**, the **--events** option to **ntracekd** command lines or via the Events tab of the Daemon Definition dialog in the **ntrace** tool.
- Invoke **ntrace** only with the trace event files that are essential to your analysis.
- Once **ntrace** is launched, select a data region of interest and discard all other events to reduce the working set size by selecting the Discard Events… option from the Events menu of a display page.
- Operate the daemons in file-wrap or buffer-wrap modes to reduce data set size in favor of keeping the most recent events.

7
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% }]
       [-x] [--nopages]
       [-u] [--use-session] [--summary=criteria]
       [--import=a.out | a.out]
       [--verbose]
       [--crash=crash_options]
       [file ...] [program_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 timeline panels (see “Timeline Panels” on page 12-1)
- displays the NightTrace Main Window (see “The NightTrace Main Window” on page 8-1)

Command-line Options

The command-line options to **ntrace** are:

-h
--help

Displays **ntrace** invocation syntax and a list of all command line options to standard output.

--help-summary

Displays help specific to the **--summary** option to standard output.

See “Summary Criteria” on page 7-6 for more information.

-v
--version

Displays the current version of NightTrace to standard output and exits.

--crash=*crash_options*

Displays available kernel trace data at the time of system crash. This option is useful if kernel tracing was running when the system crashed. It extracts kernel trace data from the in-memory kernel buffers at the time of the crash.

The crash option parameter may be either the time-date format of the crash dump under **/var/crash/save** (or **/var/kdump**) or the full paths of the namelist and vmcore files if the default crash path has been changed. For example:

--crash=08.02.06-19.11.47
--crash=/crashfiles/vmlinux-33,/crashfiles/vmcore-33.gz

The **--crash** option is only supported under Redhawk 4.1 or later and may not be available on AMD64 systems.

-l
--listing

Displays a chronological listing of all trace events and their arguments from all supplied trace-event data files to standard output and exits.

The output includes the following information about a trace event:

- relative timestamp
- trace event ID
- any trace event argument(s)
- the 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 7-11.

NOTE

The CPU field is only meaningful for kernel trace events; for user trace events, the CPU field is displayed as `CPU=??`.

--stats

Displays simple overall statistics about the trace-event data files to standard output and exits.

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:

```
ntraceud /tmp/data
```

collects trace data from any user applications which are logging the data to /tmp/data. (see “Capturing User Events with ntraceud” on page 3-1).

Issuing the command:

```
ntrace --stats /tmp/data
```

results in the output similar to the following (assuming user application were actually logging data):

```
Read 1 trace event segment timestamped with Intel TSC.
(1)  User trace event log file:  /tmp/data.
     2268 trace events saved.
     0 trace events lost.
     2.9707482s time span, from 0.0000000s to 2.9707482s.

     2268 total events read from disk.
     2268 total events saved in memory.
     0 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.

-n
--notimer

Excludes from analysis trace events for system timer interrupts in the kernel trace file.

-s *val*
--start={ *offset* | *time*{ **s** | **u** } | *percent***%** }

Excludes 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:

offset

Load trace events after the specified trace event offset.

time{ **s** | **u** }

Load trace events after the specified relative time in seconds (**s**) or microseconds (**u**).

*percent***%**

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***%** }

Excludes 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:

offset

Load trace events before the specified trace event offset.

time{ **s** | **u** }

Load trace events before the specified relative time in seconds (**s**) or microseconds (**u**).

*percent***%**

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.

-x
--notimelines

Starts NightTrace but does not include any timeline panels.

-u
--use-session

Automatically loads the last session used in a previous invocation of NightTrace. All files associated with the previous session are automatically loaded.

--summary=*criteria*

Provides a textual summary of specified trace events using the supplied *criteria*. Summary results are sent to standard output.

See "Summary Criteria" on page 7-6 for details regarding valid *criteria.*

--import=a.out
a.out

These options specify the executable file containing daemon definitions and the location of format tables and event description files. This information is embedded in executable files when they contain instrumented code generated by the NightTrace illuminator tool.

A daemon definition is created with the number of buffers, buffer length, and trace key file information extracted from the file. If the executable file does not include such information, ntrace queries the user for the name of the trace key file, and uses default values for other daemon settings.

NightTrace loads all event description and format table files gleaned from the executable.

Specifying **a.out** as a standalone argument processes executable files in the same manner as those specified with **--import**. In addition, NightTrace loads the user trace data file as specified by information embedded by the built-in "main" illuminator if it was included in the program. NightTrace also records the pathname of the specified file and associates it with any references to the base name of the file in `lookup_pc()` references during the NightTrace session. For example:

```
ntrace /tmp/a.out
```

References to "a.out" in `lookup_pc()` expressions in the session will use **/tmp/a.out** as the path to the file from which PC descriptions (routine, file and line number) are read.

--verbose

In addition to the cumulative statitistics normally output, this option provides detailed information about each occurrence of the item being summarized.

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 7-10 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 /tmp/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 provides detailed information about each occurance of the item being summarized in addition to the cumulative statitistics normally output.

This criteria consists of a comma-separated list of any of the following:

crit

This allows previously-defined profiles to be referenced when doing command line summaries.

To use previously-defined profiles when executing a summary from the command line, specify the desired profile name (*crit*) on the command line along with the NightTrace session configuration file which contains that profile

ev:*event*

Summarizes the number of occurrences of the specified *event.*

p:*process*

Summarizes all events associated with the specified *process*.

t:*thread*

Summarizes all events associated with the specified *thread*.

s:*call*

Summarizes all events associated with the entry or resumption of the specified system *call*.

sl:*call*

Summarizes all events associated with the exit or suspension of the specified system *call*.

se:*call*

Summarizes all events associated with the specified system *call*.

ss:*call*

Summarizes all occurrences of a state defined by system call activity for the specified system *call*.

i:*intr*

Summarizes all events associated with the entry or resumption of the specified interrupt *intr*.

il:*intr*

Summarizes all events associated with the exit or interruption of the specified interrupt *intr*.

ie:*intr*

Summarizes all events associated with the specified interrupt *intr*.

is:*intr*

Summarizes all occurrences of a state defined by interrupt activity for the specified interrupt *intr*.

e:*exc*

Summarizes all events associated with the entry or resumption of the specified exception *exc*.

el:*exc*

Summarizes all events associated with the exit or interruption of the specified exception *exc*.

ee:*exc*

Summarizes all events associated with the specified exception *exc*.

es:*exc*

Summarizes 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*

Summarizes 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 in a comma-separated list.

Consider the following example:

```
ntrace --summary=ev:5,ss:read,ss:alarm,crit_0 event_file my_session
```

Using the trace event file **event_file** as the trace data source (see "Trace Event Files" on page 7-11), NightTrace will:

1. 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`)
2. 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`)
3. perform a summary using the profile defined by `crit_0` in the **my_session** session file (see "Session Configuration Files" on page 7-24)

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 7-24).

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 NightTrace. 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 7-11 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 7-11 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 profiles 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 7-24 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 Trace Segments... menu choice of the File menu on the NightTrace Main Window.

  See "Trace Data Segments" on page 7-25 for more information.

- program file

  Application Illumination embeds in executable object files paths to various support files that **`ntrace`** can extract:

  - event map files defining names for the events generated for function entry and return points;
  - configuration files containing format tables to neatly format the events and their arguments generated for function entry and return points;

a trace event file if the `main` illuminator is used (this file may be recorded using a relative path; if this is the case, ntrace must be invoked with the same current working directory that the program file was executed with).

See "Application Illumination" on page 5-1 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, or
- select the Open Files... menu option from the File menu of the NightTrace main window 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, using these files 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
- select the Open Files... menu item from the File menu on the NightTrace Main Window

You can create an event map file with a text editor before you invoke NightTrace.

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 underscores. 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 profile referring to those events the same name in lowercase. For more information about profiles of events, see "Profile References" on page 16-193.

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:

`%d`

signed decimal integer (default)

`%o`

unsigned octal integer

`%x`

unsigned hexadecimal integer

`%lf`

signed double precision, decimal floating point

For more information on these conversion specifications, see **`printf(3)`**.

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 Strings Tables" on page 7-17.

Table Files

A *table file* contains information used to obtain verbose descriptions of events or arguments associated with events..

A table file is an ASCII file containing such definitions as:

- string table definitions (see "String Tables" on page 7-15)
- format table definitions (see "Format Tables" on page 7-20)

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.

To load an existing table file, either:

- specify the configuration file as an argument to the **`ntrace`** command when you invoke NightTrace
- Select the Open Files... menu option from the NightTrace menu of the NightTrace Main window and select the configuration file from the file selection dialog

Tables

The table file may contain two types of tables, both of which can improve the readability of your NightTrace displays:

- string tables (see "String Tables" on page 7-15)
- format tables (see "Format Tables" on page 7-20)

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`*

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 Tables" on page 7-17, and page 17-17.

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 16-184.

Pre-Defined Strings Tables

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, 4306)
get_item(event, "IRQ_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")
```

`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.

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")
```

`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 mutipile 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 16-55 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_sbc1, 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 17-17.

`syscall`

See page 17-17.

`device`

See page 17-17.

`vector_`*`nodename`*

See page 17-17.

`syscall_`*`nodename`*

See page 17-17.

`device_`*nodename*

See page 17-17.

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.

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 ) = {
    [ index_type = "event"; ]
    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 *index_type* of "*event*" may be specified to direct ntrace to use this table to format events and their arguments. More than one table may have the *event* index_type.

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:

`%i`

Signed integer

`%u`

Unsigned decimal integer

`%d`

Signed decimal integer

`%o`

Unsigned octal integer

`%x`

Unsigned hexadecimal integer

`%lf`

Signed double precision, decimal floating point

`%e`

Signed decimal floating point, exponential notation

`%c`

Single character

`%s`

Character string

`%%`

Percent sign

`\n`

Newline

For more information on these conversion specifications, see **`printf(3)`**.

format_string may contain any number of conversion specifications. There is a one-to-one correspondence between conversion specifications and quoted val-

ues. 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 16-184). For more information on expressions, see "Using Expressions" on page 16-1.

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 16-184.

For more information on format tables and the `get_format()` function, see "get_format()" on page 16-188.

For more information about `arg1()`, see "arg()" on page 16-21.

For more information about `offset()`, see "offset()" on page 16-49.

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 7-5) is strongly encouraged to invoke NightTrace with the last session loaded or saved.

A session configuration may include:

- daemon definitions

  See "Edit Daemon Definition" on page 9-8 for more information.

- display page configurations

  See "Table Files" on page 7-14 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

- profiles of conditions and states

  See "Using Expressions" on page 16-1 for more information.

- named tags

  See "Tags List Panel" on page 15-1 for more information.

- previously-executed searches
- previously-executed summaries
- references to saved trace data segment files

  See "Trace Data Segments" on page 7-25 for more information.

- references to kernel trace files generated by **`ntracekd`** (see "The ntracekd Daemon" on page 4-1), or a kernel daemon defined in the GUI (see "Daemons Panel" on page 9-1)

- references to user trace files generated by **`ntraceud`** (see “The ntraceud Daemon” on page 3-1), or a user daemon defined in the GUI (see “Daemons Panel” on page 9-1)

Session configuration files can be generated by the following menu items in the File menu of the NightTrace Main Window:

Upon exiting when there are unsaved changes to the session, the user is given the chance to save the changes before NightTrace exits.

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 7-1)
- using the Load Session dialog 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 Trace Data button on the Trace Segments panel (see “Trace Segments Panel” on page 10-1 for more information).

8
The NightTrace Main Window

The NightTrace GUI is invoked using **`ntrace`** (see “Invoking NightTrace” on page 7-1).

By default, the NightTrace main window is presented as shown in the figure below.

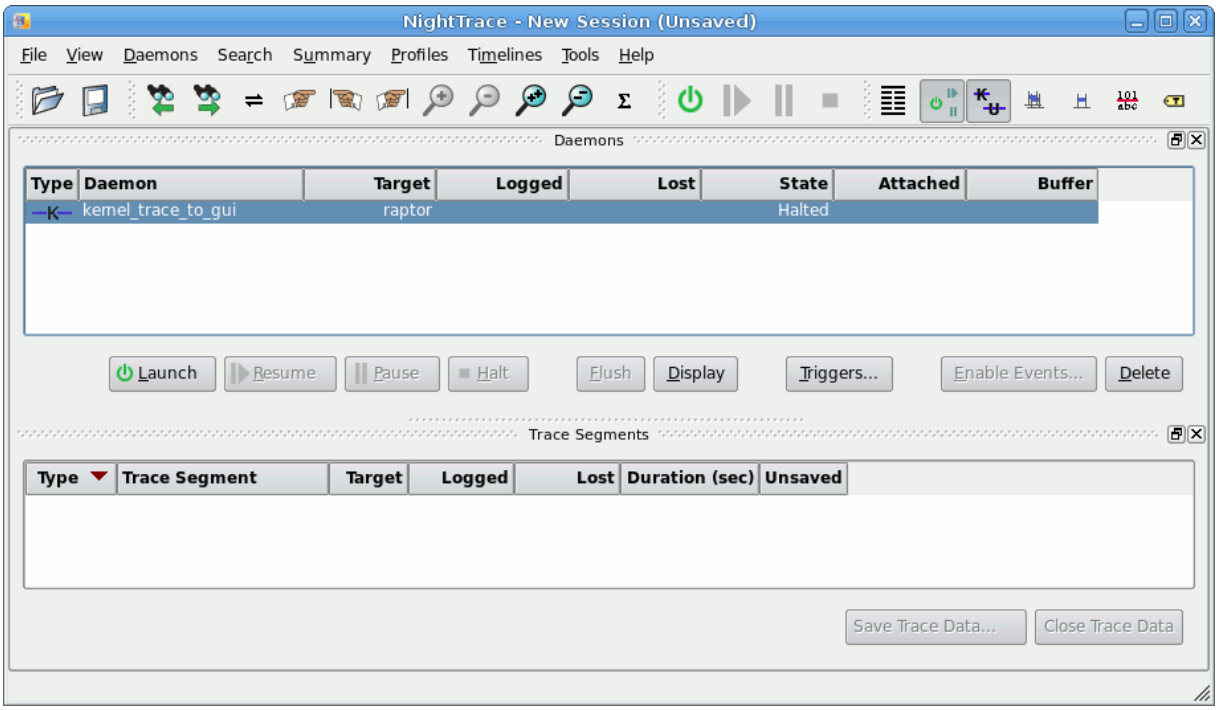


Figure 8-1. NightTrace Main Window

The NightTrace main window consists of the following components:

- Menu Bar
- Toolbars
- Pages and Panels

Menu Bar

The menu bar provides access to session configuration services, additional tools, and help. The menu bar provides the following menus:

- File
- View
- Daemons
- Search
- Summary
- Profiles
- Timelines
- Tools
- Help

Each menu is described in the sections that follow.

File

Accelerator: Alt+F

The File menu contains session-related items such as initiating a new *session*, saving the current session, and opening a previously-saved session or data file.

A session includes daemon configurations, trace data sets, configuration options, display pages, and user-defined profiles.

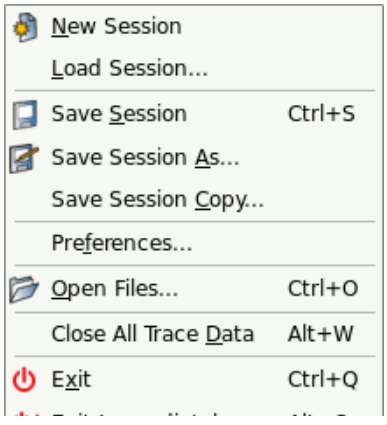


Figure 8-2. File Menu

The following paragraphs describe the options on the File menu in more detail.

New Session

Mnemonic: N

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 session but have not yet been saved, NightTrace will ask you if you wish to save the current session before proceeding.

Load Session...

Mnemonic: L

This option launches a standard file selection dialog which allows you to specify a previously-saved session file. Filenames displayed in the file selection dialog are relative to the host system.

If changes have been made to the current session but have not yet been saved, NightTrace will ask you if you wish to save the current session before proceeding.

NOTE

NightTrace will automatically load the last session used when invoked with the **-u** option. See “Invoking NightTrace” on page 7-1 for more information.

Save Session

Mnemonic: S
Accelerator: Ctrl+S

Save Session saves the current session to a session configuration file.

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 window title.

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. 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 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

If the display pages were 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 As...

Mnemonic: A

This option launches a standard file selection dialog which allows you to specify the a filename where the session will be saved. Filenames displayed in the file selection dialog are relative to the host system.

Save Session Copy

Mnemonic: C

Save Session Copy saves the current session to a newly created session configuration file (see "Session Configuration Files" on page 7-24 for a complete description of the contents of a session).

In addition, all trace data and display pages are saved to new file names using a common session file name prefix.

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.

Preferences...

Mnemonic: F

This option launches the Preferences Dialog which allows you to specify preferences for NightTrace, including font selection.

Saved user preferences are applied to all NightTrace invocations for the user. Preferences are saved in the user's home directory and have a broader application than session configuration files.

See "Preferences Dialog" on page 8-40 for more information.

Open Files...

Mnemonic: O
Accelerator: Ctrl+O

Presents the user with a standard file selection dialog so that they may select a trace event file, event map file, or configuration file to load.

The trace event file can be a user trace data file or a kernel trace data file. See "Trace Event Files" on page 7-11 for more information.

An event map file provides ASCII names for specific trace event values. See "Event Map Files" on page 7-11 for more information.

Configuration files contain string and format tables as well as display page definitions. See "Table Files" on page 7-14 for more information.

Close All Trace Data

Mnemonic: D
Accelerator: Alt+W

Closes the trace data segments currently selected in the Trace Segments area. The events associated with the closed segments are immediately removed from the current data set being analyzed.

Data segments that were not associated with a trace file and that have not yet been saved will be lost when closed.

Exit

Mnemonic: X
Accelerator: Ctrl+Q

Closes the session and exits NightTrace completely.

If changes have been made to the current configuration but have not yet been saved, NightTrace will ask you if you wish to save the session before proceeding.

Exit Immediately

Mnemonic: I
Accelerator: Alt+Q

Closes the session and exits NightTrace without prompting to save changes that have been made. Any changes will be lost.

View

Accelerator: Alt+V

The View menu allows you to add, rename, or delete pages and controls which panels in pages are visible.

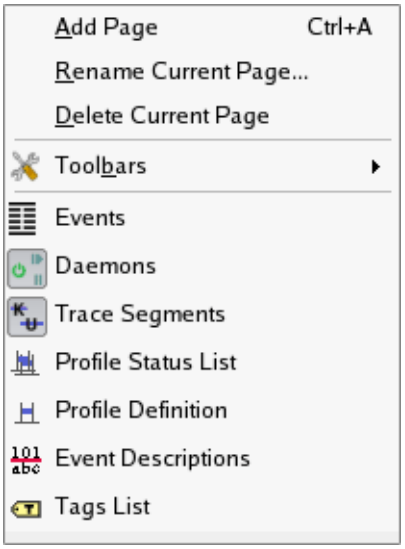


Figure 8-3. View Menu

Add Page

Mnemonic: A
Accelerator: Ctrl+A

This option adds a new page to the right of the last page in the main window.

Rename Current Page...

Mnemonic: R

This option launches a dialog that allows you to change the name of the current page. The current page is the page which is currently being displayed in the main window.

This option is also available from the context menu which appears when you right-click on a page's tab.

Delete Current Page

Mnemonic: D

This option deletes the current page and all panels it contains. The current page is the page which is currently being displayed in the main window.

This option is also available from the context menu which appears when you right-click on a pages's tab.

Toolbars

Mnemonic: B

Figure 8-4. Toolbars Menu

This menu allows you to hide or show individual Toolbars on the main window. You can also hide or show toolbars using the context menu that appears when you right-click a toolbar.

Events

This checkbox controls whether the Events panel is displayed. See "Events Panel" on page 11-1 for information its operation.

Daemons

This checkbox controls whether the Daemons panel is displayed. See "Daemons Panel" on page 9-1 for information on its operation.

Trace Segments

This checkbox controls whether Trace Segments panel is displayed. See "Trace Segments Panel" on page 10-1 for information on its operation.

Profile Status List

This checkbox controls whether the Profile Status List panel is displayed. See "Profile Status List Panel" on page 13-12 for information on its operation.

Profile Definition

This checkbox controls whether the Profile Definition panel is displayed. See "Profile Definition Panel" on page 13-1 for information on its operation.

Event Descriptions

This checkbox controls whether the Event Descriptions panel is displayed. See "Event Descriptions Panel" on page 14-1 for information on its operation.

Tags List

This checkbox controls whether the Tags List panel is displayed. See "Tags List Panel" on page 15-1 for information on its operation.

Timelines and Panels

When timelines or other panels are added, an entry for each is added to the View menu. These entries are checkboxes which toggle the visibility of the panel in the current page.

Daemons

Accelerator: Alt+D

The Daemons menu provides functionality for configuring new and existing daemon definitions, as well as attaching to and detaching from running daemons.

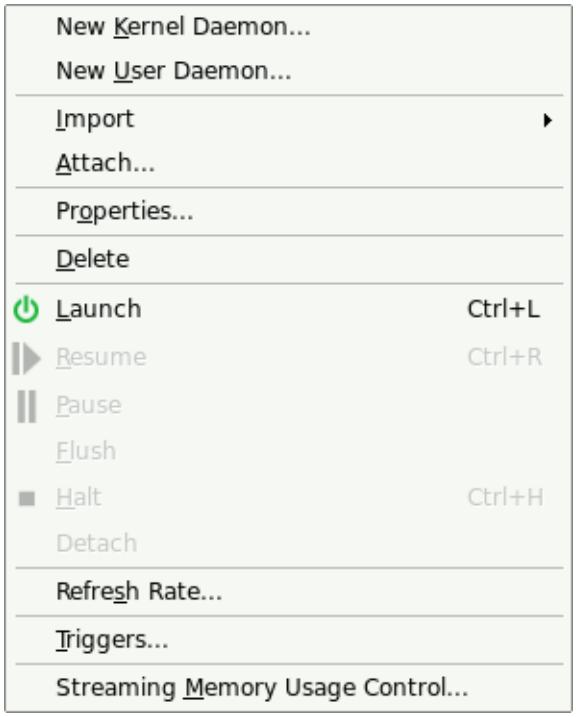


Figure 8-5. Daemons Menu

This menu is identical to the context menu shown when right-clicking inside the Daemons panel, as described in “Daemons Panel” on page 9-1.

Search

Accelerator: Alt+R

The Search menu contains search-related items such as opening the Profile Definition panel to define search criteria, executing a forward or backward search with the most recent search criteria, or modifying search options.

Figure 8-6. Search Menu

Text Search

This option launches the Search Events for Text dialog which allows you to specify textual search criteria for searching the contents of an Events panel. See “Text Search” on page 11-3 for a description of this dialog and its actions.

Change Search Profile...

Mnemonic: S
Accelerator: Ctrl+F

Displays a Profiles Definition panel allowing you to define the search criteria and to execute a search for an event or condition in a Timeline panel. See “Profile Definition Panel” on page 13-1 for more information.

If a Profiles Definition panel already exists on a page, that page is raised; otherwise, a new page is created that contains a Profiles Definition panel.

Search Forward

Mnemonic: R
Accelerator: Ctrl+G

Executes a forward search using the last profile defined or selected. If no profiles have been defined, a forward search for the next event is executed.

Search Backward

Mnemonic: K
Accelerator: Ctrl+B

Executes a backward search using the last profile defined or selected. If no profiles have been defined, a backward search for the previous event is executed.

Search Forward withinTimeline Interval

Accelerator: Alt+G

Executes a forward search using the last profile defined or selected. If no profiles have been defined, a forward search for the next event is executed. The search is bounded by the events in the current timeline interval.

Search Backward within Timeline Interval

Accelerator: Alt+B

Executes a backward search using the last profile defined or selected. If no profiles have been defined, a backward search for the previous event is executed. The search is bounded by the events in the current timeline interval.

Halt Search

This option terminates an active search and leaves the current timeline unchanged.

Goto Next Tag
Goto Previous Tag

Mnemonics:] and [

These options search forward or backward, respectively, to the next or previous tagged event or time in the data set.

Go Back to Previous Interval

Accelerator: Ctrl+V

This option toggles the current timeline between its current position and its last position. Using this option or accelerator, you can easily revert back to a location in the data set after executing a search or clicking elsewhere in a timeline or ruler.

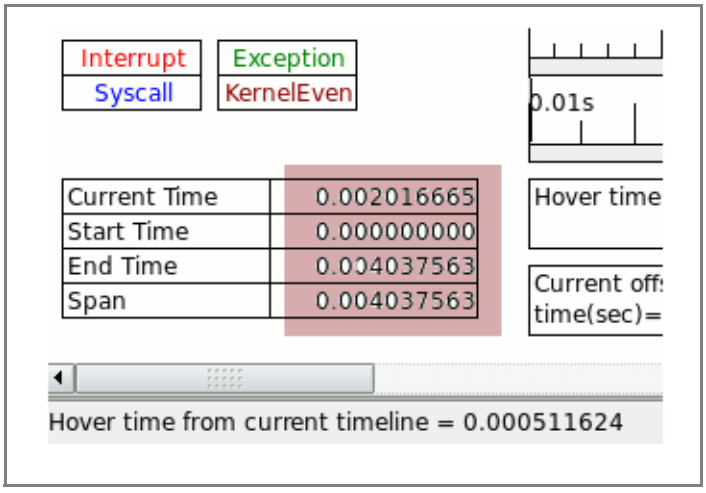
Goto...

Mnemonic: G
Accelerator: Ctrl+I

This option launches the Change Interval dialog which allows you to change the current time and boundaries of the current interval.

Figure 8-7. Change Interval Dialog

The Change Interval dialog is launched from the Goto… option of the Search menu. It is also launched whenever you click on any of the values in the interval value boxes in the lower-left corner of a timeline,



as shown in the picture above (highlighted with a reddish background).

The dialog allows you to enter values as event offsets or times. Values entered in floating-point notation are interpreted as times, as are values with a trailing s character (meaning seconds). Integer values without a trailing s character are interpreted as event offsets.

In most situations, you should change at most one or two of the values in the dialog and let NightTrace adjust the unmodified values for you when you press OK in order to accommodate your specifications.

For example, if you simply change the Interval End setting to a larger number, NightTrace will expand the Interval Span (and change the Current timeline value if necessary) when you press OK.

The dialog was designed for quick access and use. For example, to change the current timeline to time 3.5s, you could use the following 6 keystrokes when a timeline panel has focus (the keystrokes are separated by whitespace for clarity below):

Ctrl+I 3 . 5 s Enter

When the dialog is launched via the menu or accelerator sequence, the Current time value is fully selected so that it will be replaced immediately with whatever characters you type. The OK button has the activation focus, so that hitting the Enter key activates the OK button.

When the dialog is launched by clicking on one of the actual values that define the interval in the lower-left corner of a timeline (see picture above), the value that you clicked on is fully selected in the dialog, ready for immediate substitution.

Goto First Event

Mnemonic: F
Accelerator: Alt+LeftArrow

This option searches to the first event in the data set.

Goto Last Event

Mnemonic: L
Accelerator: Alt+Right

This option searches to the last event in the data set.

Ask Before Wrapping for Search

When checked, this causes a dialog to pop up when either end of the data set is reached during a search operation; it allows you to continue searching at the other end or to cancel the search.

Zoom to Search Match

When checked and a search criteria is found, the timeline is zoomed to include the number of events specified by the Limit Number of Events Displayed... option of the Timelines menu.

Summary

Accelerator: Alt+U

The Summary menu provides for defining profiles for summaries, executing summaries, and controlling summary options.

Figure 8-8. Summary Menu

Change Summary Profile...

Mnemonic: U
Accelerator: Ctrl+U

This option opens the Profile Status List and Profiles Definition panels allowing you to select a profile to summarize or define a new profile to summarize. If these panels already exist on a page, that page is raised; otherwise a new page is added which contains these panels. See “Profiles Panels” on page 13-1 for more information.

Summarize

Mnemonic: Z
Accelerator: Ctrl+Z

This option executes a summary on the current profile. If no profiles have been defined, a summary of all events is executed. For each summary of a specific profile, a new page is created to hold the summary results, including any required data graphs as directed by the Graph State Durations... or Graph State Gaps... options of the Summary menu.

Summarize within Timeline Interval

Mnemonic: I
Accelerator: Alt+Z

This option is identical to the Summarize option except that the list of events to summary is constrained by those in the current timeline interval.

Graph State Durations...

Mnemonic: D

This option displays the Graph State Durations dialog which allows you to select whether you want a data graph generated when summarizing the current profile. The data graph shows the individual durations of each instance of the state as defined by the profile, plotted vertically.

The dialog also allows you to specify a standard deviation value which instructs the summary action to graph values that fall outside the specified domain as the maximum defined by that domain.

Graph State Gaps...

Mnemonic: G

This option is identical to the Graph State Durations option except that it controls the graphing of the gaps between instances of states as defined by the current profile.

Prevents the current timeline from being moved, but the summary results are still displayed in page text areas.

Profiles

Accelerator: Alt+P

The Profiles menu manipulates the list of profiles shown in the Profile Status List panel.

A profile is a set of criteria either defining a state with beginning and end conditions, or simply a condition. Profiles are used for searches, summaries, and graphs.

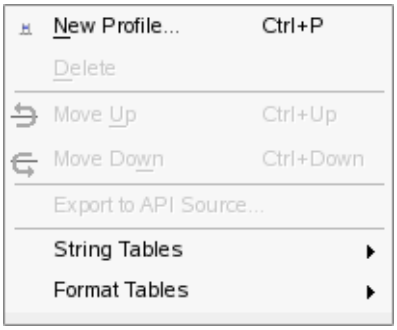


Figure 8-9. Profiles Menu

New Profile...

Mnemonic: N
Accelerator: Ctrl+P

This option shows the Profile Status List and Profile Definition panels. If these panels already exist on a page, the page is raised; otherwise, a new page is created which contains these panels. See “Profiles Panels” on page 13-1 for more information on using profiles.

Delete

Mnemonic: D

This menu choice deletes all profiles currently selected in the Profile Status List panel.

Move Up
Move Down

Accelerator: Ctrl+UpArrow and Ctrl+DownArrow

These options move the currently selected profiles in the Profile Status List panel towards the beginning or end of the list, respectively.

Export to API Source...

This option opens the Export Profiles to NightTrace API Source File dialog to automatically generate source code defining and referencing profiles, for use with applications using the NightTrace Analysis API (see “Using the NightTrace Analysis API” on page 18-1).

String Tables

This option expands to a sub-menu which allows you to select an existing string table for modification, or to create a new string table.

Format Tables

This option expands to a sub-menu which allows you to select an existing format table for modification, or to create a new format table.

Export Profiles to NightTrace API Source File

The Export Profiles to NightTrace API Source File dialog is presented when the Export to API Source... menu item is selected from the Profiles menu.

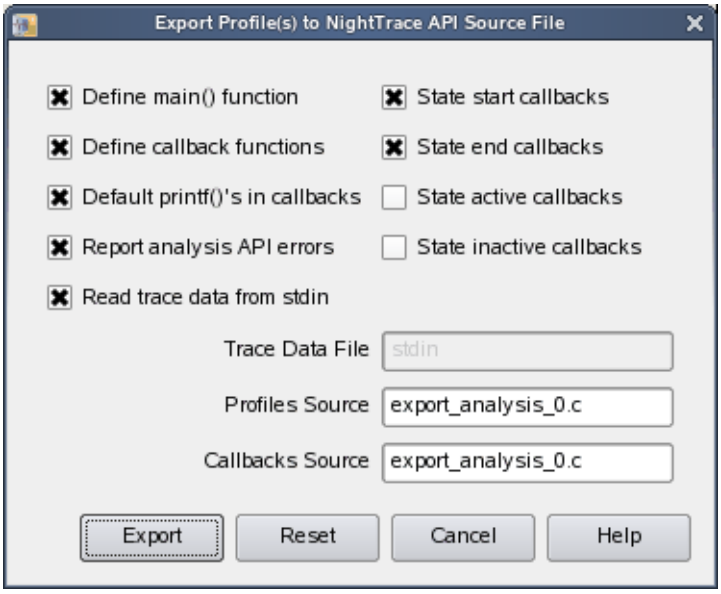


Figure 8-10. Export Profiles Dialog

This dialogs generates C source code using the NightTrace Analysis API to define and install listener callback functions for the profiles selected from the Profile Status List panel when the dialog was launched.

Define main() function

When checked, this option generates source code for a main C program which creates an instance of the Analysis API and installs all definitions and callbacks selected in this dialog.

Define callback functions

When checked, this option generates stub routines for all callback functions that are defined by this dialog. The stub routines are empty unless the Include default printf() output in callbacks option is checked. If this option is not checked, the function profiles are still generated, but no definitions are generated.

Default printf()'s in callbacks

When checked, this option generates source code to print information about instances of the selected profiles in the callback function definitions.

Report analysis API errors

When checked, this function will report all errors from API calls to `stderr`; otherwise, errors are ignored.

Read trace data from stdin

This option controls the initial API calls which either open a pre-existing data file or read data from `stdin` in streaming mode.

State start callbacks

When checked, a callback profile is generated and registered with the API for the start event of the selected state profiles.

State end callbacks

When checked, a callback profile is generated and registered with the API for the end event of the selected state profiles.

State active callbacks

When checked, a callback profile is generated and registered with the API for any event that occurs when selected state profiles are active.

State inactive callbacks

When checked, a callback profile is generated and registered with the API for any event that occurs when selected state profiles are inactive.

Trace Data File

When Read trace data from stdin is not checked, this text field defined the data file from which pre-existing data will be read.

Profiles Source

This text area defines the name of the source file for all source code generated except for callback definitions.

Callbacks Source

This text area defines the name of the source file for all source code that define callback routines.

By default, the dialog is set to create a fully functional program that you can compile and link using a command similar to the following:

```
cc export_analysis_0.c -lntrace_analysis
```

You could subsequently feed live NightTrace data to the program using an invocation similar to the following:

```
ntraceud --stream /tmp/key-file | ./a.out
```

See "Using the NightTrace Analysis API" on page 18-1 for more information.

Timelines

Accelerator: Alt+M

The Timelines menu allows to create new timeline panels and provides controls for moving and changing timeline intervals.

Figure 8-11. Timelines Menu

The Timelines menu in the main window menu bar is essentially identical to the context menu available from all Timeline panels, with the addition of the New submenu which allows you to create new timelines.

This section will describe the New sub-menu; see "Timeline Panels" on page 12-1 for a description of the remaining menu items.

New

Mnemonic: N

Empty Timeline

Mnemonic: T

This menu choice opens a new timeline so that the user may configure it from scratch. The grid must be populated with display objects before trace information can be analyzed or graphically examined. See "Timeline Panels" on page 12-1.

Default User Timeline

Mnemonic: U

This menu choice opens the default user timeline which is automatically pre-configured to show all user events and specific descriptions of the event ID and the first argument of each event.

The default user timeline includes a row that includes events for each registered thread in the application, as well as a row that includes events for all threads.

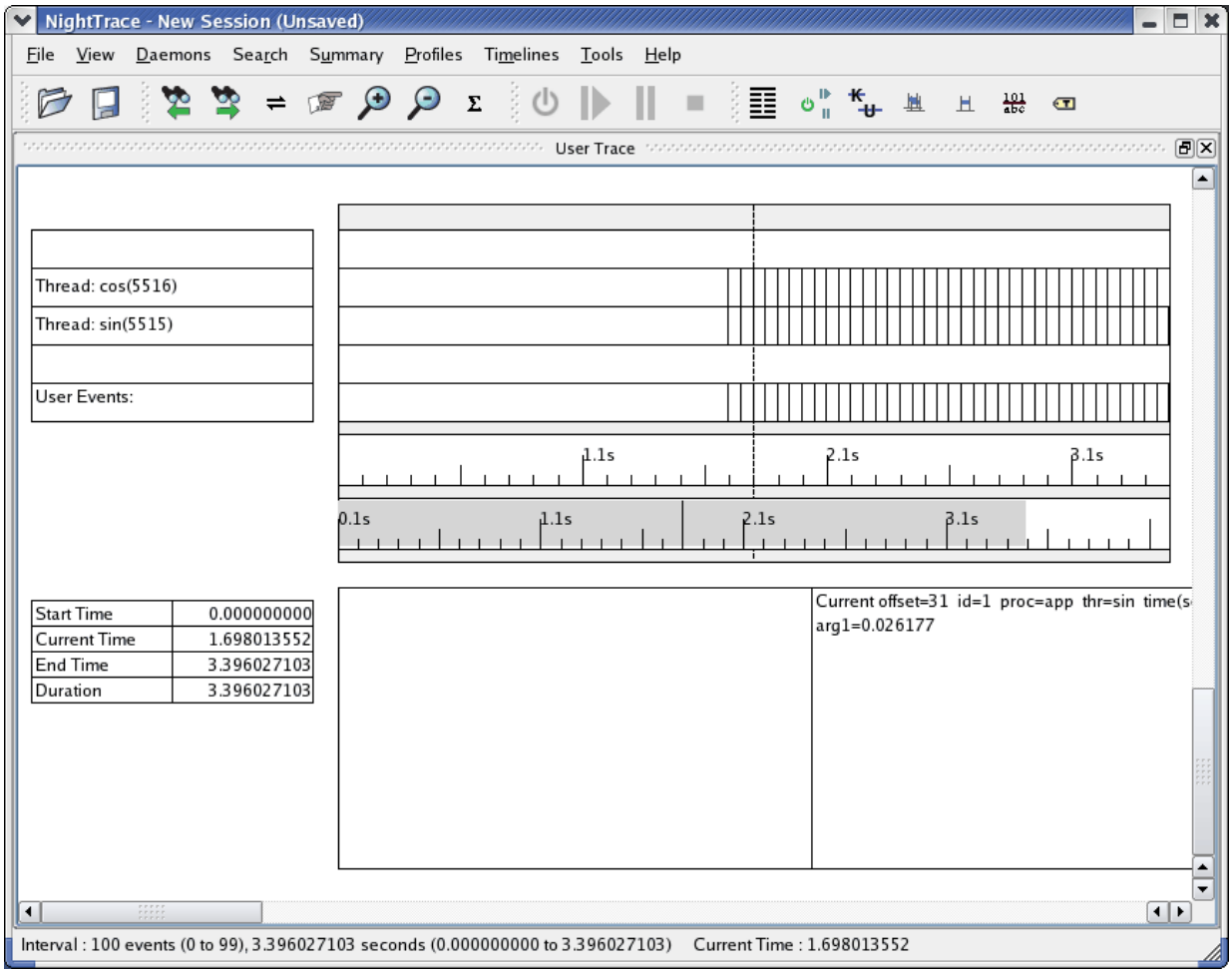


Figure 8-12. Default User Timeline

Default Ada Timeline

Mnemonic: A

This menu choice builds a user timeline which is automatically configured to show *task-information* displays for every Ada task in the current trace data set.

A task-information display includes the following information: the task name, the pid and Ada task ID, and a state graph indicating various Ada language events and states, especially as related to tasking and exceptions.

Default AI Timeline...

Mnemonic: I

This menu choice opens the default Application Illumination timeline. This is essentially a single-thread version of the default timeline, but with bigger hover and descriptive areas, as Application Illumination event descriptions tend to be verbose.

See "Application Illumination" on page 5-1 for more information.

Custom Kernel Timeline...

Mnemonic: K

Presents the Build Custom Kernel Page dialog 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

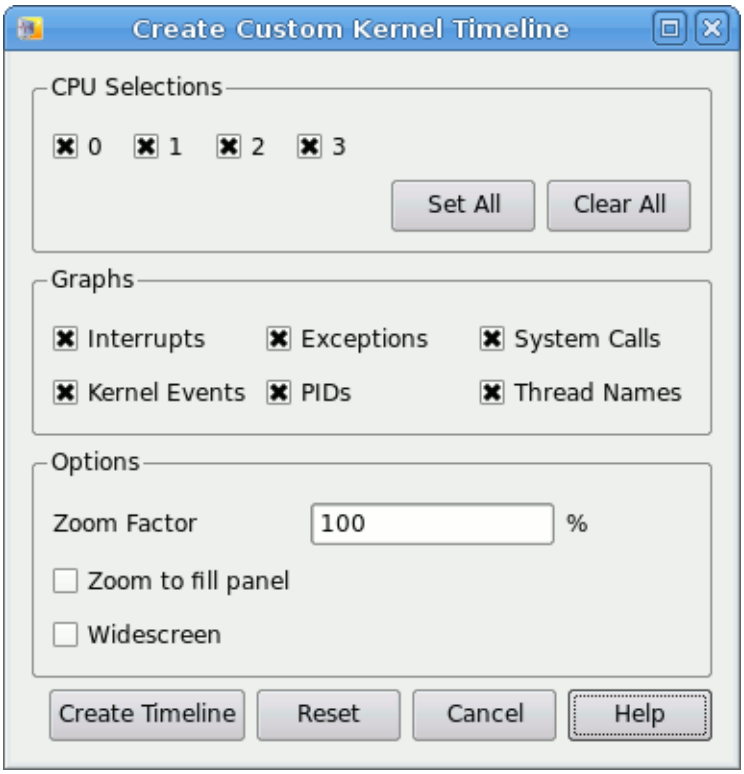


Figure 8-13. Create Custom Kernel Timeline Dialog

The checkboxes allow you to select which event and state graphs you wish to build for which CPUs.

The Options area of the dialog provides additional tailoring choices, especially useful when you have 8 or more CPUs to view.

The Zoom Factor specifies the percentage of the default kernel timeline geometry that should be used in creating the new page. For most monitors, a kernel timeline for 8 CPUs doesn't fit vertically in the visible area of the display (although the timeline does have a scroll bar so you can scroll to see all CPUs). Selecting a percentage less than 100% may be useful in such a situation.

The Zoom to fill panel checkbox tells NightTrace to automatically calculate the Zoom Factor so that all the CPUs will fit in the available vertical space of screen. NightTrace calculates the available space based on the current size of the NightTrace main window. For best results, increase the size of the NightTrace main window (or maximize the window to fill the entire screen) before creating the custom timeline.

The Zoom to fill panel checkbox disables and overrides the Zoom Factor setting, because it automatically calculates the setting when creating the timeline.

The Widescreen checkbox splits the kernel timeline in half, creating two columns of CPUs. You can select the Widescreen option as well as a zoom option.

The Zoom to fill panel may create a timeline in which the labels are effectively unreadable. If this occurs, try also selecting Widescreen and/or excluding CPUs or individual graphs from the dialog.

See "Kernel Tracing" on page 17-1 for more information.

Per Process Kernel Timeline...

Mnemonic: P

Presents a list of processes in the current kernel data set which allows you to quickly build a customized kernel timeline that is filtered to display specific processes.

NOTE

Support for kernel tracing is only available under some operating system distributions. See "Kernel Dependencies" on page B-1 for more information.

Limit Number of Events Displayed...

This option launches a simple dialog which allows you to set a display limit, in units of events. This limit is consulted when doing search operations and when using the Zoom In to Limit action.

Zoom

Mnemonic: Z

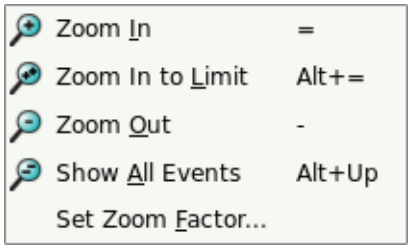


Figure 8-14. Zoom sub-menu of Timelines Menu

Zoom In

Mnemonic: I
Accelerator: =

Change the current interval such that fewer events are displayed, but with more detail.

The amount the interval changes is dependent on whether or not you have selected events in a timeline.

If you use the mouse to click-and-drag to select events in a timeline, then the Zoom In action will change the interval to include only the events you have selected.

Otherwise, a Zoom In action will change the interval by the Zoom Factor.

Zoom In to Limit

Mnemonic: L
Accelerator: Alt+Down, Alt+=

Change the current interval by zooming in to the smallest interval as defined by the Limit Number of Events Displayed... menu setting described above.

Zoom Out

Mnemonic: O
Accelerator: -

Change the current interval such that more events are displayed, but with less detail. The change in interval is controlled by the Zoom Factor.

Show All Events

Mnemonic: A
Accelerator: Alt+Up

Change the current interval by zooming all the way out such that the interval contains the entire data set. When fully zoomed out, the display isn't useful for detailed analysis, but it is useful for identifying areas of significant activity, etc.

Set Zoom Factor...

Mnemonic: F

This menu option launches a simple dialog which allows you to change the Zoom Factor. The Zoom Factor is a floating point number which represents the change in interval when incremental zoom actions are taken.

Thus a zoom factor of 2.0 will cause roughly twice as long an interval to be displayed after a single zoom out action, and 1/2 as long an interval to be displayed after a zoom in action.

Shift Left

Mnemonic: L
Accelerator: Ctrl+Left

Shift the current interval "left", so that the interval now includes earlier times. The amount the interval changes is controlled by the Interval Shift setting, which you can set via the Shift Percentage… menu option.

By default, the Interval Shift setting is 25%, so that when you shift an interval left (or right), the new interval still includes 75% of the time covered by the previous interval. This can be helpful when you want to maintain some context while traversing the data set.

Shift Right

Mnemonic: R
Accelerator: Ctrl+Right

Shift the current interval "right", so that the interval now includes later times. The amount the interval changes is controlled by the Interval Shift setting, which you can set via the Shift Percentage… menu option.

By default, the Interval Shift setting is 25%, so that when you shift an interval right (or left), the new interval still includes 75% of the time covered by the previous interval. This can be helpful when you want to maintain some context while traversing the data set.

Shift Percentage...

Mnemonic: P

This option launches a simple dialog which allows you to change the Interval Shift value. The Interval Shift is a percentage that controls how much the interval changes when you do Shift Left or Shift Right interval options (as described above).

Setting the percentage to 25% will maintain 75% of the current interval's timespan in the new interval. This is useful when you want to maintain some context from the previous view while traversing the data set.

Setting the percentage to 100% will present an entirely new timespan for the next interval (contiguous with the previous interval).

Center Current Timeline

Mnemonic: C
Accelerator: =

This option adjusts the interval such that the current timeline is centered in the interval.

This option has no effect if there are insufficient events outside the current interval to accommodate the current Interval Span setting. In such circumstances, you should Zoom In sufficiently before selecting this option.

Discard Selected...

Mnemonic S

This option discards all events from the data set that are currently selected in the timeline (selection is done using click-and-drag operations with the mouse).

A verification dialog is presented before the events are discarded.

This option is most useful when you have a very large data set and want to concentrate on a small portion of the data and use selection to identify events you want to delete.

In such circumstances, it may be useful to save a copy of your session using the Save Session Copy... option from the File menu before discarding events. The Save Session Copy... option creates a copy of all your session information as well as all the current trace data; thus you can easily revert back to the original data set subsequently.

Discard Unselected...

Mnemonic: U

This option discards all events from the data set that are not currently selected in the timeline (selection is done using click-and-drag operations with the mouse).

A verification dialog is presented before the events are discarded.

This option is most useful when you have a very large data set and want to concentrate on a small portion of the data.

In such circumstances, it may be useful to save a copy of your session using the Save Session Copy... option from the File menu before discarding events. The Save Session Copy... option creates a copy of all your session information as well as all the current trace data; thus you can easily revert back to the original data set subsequently.

Distinguish Process Name By PID

Mnemonic: D

This option causes the process names shown in timelines to be distinguished from other processes by appending the Process ID to the name.

Edit Current Event Description...

Mnemonic: E
Accelerator: Ctrl+D

This option launches the Edit Event Description dialog which allows you to define or change the name of an event and its description. See “Edit Current Event Description...” on page 11-5 for a description of that dialog.

Tools

Mnemonic: Alt+L

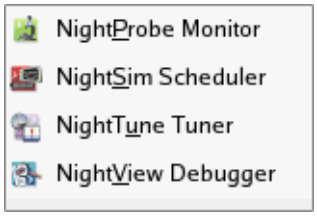


Figure 8-15. Tools Menu

The following describe the options on the Tools menu:

NightProbe Monitor

Mnemonic: P

Opens the NightProbe Data Monitoring tool. 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 for data capture or to create a "control panel" for program input and output.

NightSim Scheduler

Mnemonic: S

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.

NOTE

NightSim is not available on some systems. NightSim depends on the Frequency Based Scheduler. See "Kernel Dependencies" on page B-1 for more information.

NightTune Tuner

Mnemonic: U

Opens the NightTune Tuner. NightTune is a graphical tool for analyzing the status of the system in terms of processes, interrupts, context switches, interrupt CPU affinity, processor shielding and hyper-threading control as well as network and disk

activity. NightTune can adjust the scheduling attributes of individual or groups of processes, including priority, policy, and CPU affinity.

For systems that support CPU shielding, NightTune provides a handy interface for controlling shielding, including downing sibling hyper-threaded CPUs to avoid interference.

NightView Debugger

Mnemonic: V

Opens the NightView Source-Level Debugger. NightView is a graphical source-level debugging and monitoring tool specifically designed for real-time applications and multi-threaded applications. NightView can monitor, debug, and patch multiple real-time processes running on multiple processors with minimal intrusion.

Help

Mnemonic: Alt+H

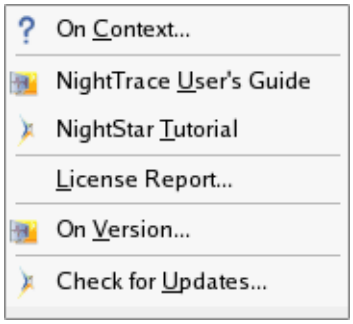


Figure 8-16. Help Menu

The following describe the options on the Help menu:

On Context

Mnemonic: C

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 Context menu item is selected). The cursor turns to the a circle with a backslash when the item under the cursor has no help description associated with it.

In addition, context-sensitive help may be obtained for the currently highlighted option by pressing the F1 key. NightStar's online help system, will open with the appropriate topic displayed.

NightTrace User's Guide

Mnemonic: G

Opens the online version of the *NightTraceRT User's Guide* in the online help viewer.

NightStar RT Tutorial

Mnemonic: T

Opens the online version of the *NightStar RT Tutorial* in the online help viewer.

License Report

Mnemonic: T

Opens a license dialog which indicates the current license server and the number of licenses available on the system.

On Version

Mnemonic: V

Displays a short description of the current version of NightTrace.

Check for Updates...

Mnemonic: U

Launches NUU (Network Update Utility) enabling you to update your system with the latest NightStar software. This requires network access to Concurrent's Updates web site. Updates require a login and user ID issued by Concurrent. Refer to http://redhawk.ccur.com/updates for complete information.

Toolbars

NightTrace includes four toolbars which can be dragged and placed on any corner or side of the main window. These include:

- the File Toolbar
- the Search Toolbar
- the Daemons Toolbar
- the Panels Toolbar

File Toolbar

This toolbar consists of two icons.

Open Files

When pressed, this icon invokes the action associated with the Open Files.... option of the File menu.

Save Session

When pressed, this icon invokes the action associated with the Save Session option of the File menu. This icon is disabled if no changes have been made to the current session since it was last loaded or saved.

Search Toolbar

This toolbar consists of seven icons.

Search Backward

When pressed, this icon searches backward in the data set from the current timeline for the nearest occurrence of the profile selected in the Profile Status List panel. If no profile is selected, it searches backward for the nearest event.

Search Forward

When pressed, this icon searches forward in the data set from the current timeline for the nearest occurrence of the profile selected in the Profile Status List panel. If no profile is selected, it searches forward for the nearest event.

Go Back To Previous Interval

When pressed, this icon invokes the Go Back to Previous Interval option of the Search menu, allowing you to switch back and forth between the current timeline and the last value of the current timeline.

Goto

When pressed, this icon invokes the Goto… option of the Search menu, allowing you to type in an event offset or time of interest.

Goto First Event

When pressed, this icon changes the current timeline to be the first event in the data set.

Goto Last Event

When pressed, this icon changes the current timeline to be the last event in the data set.

Zoom In

When pressed, this icon causes the time interval to be reduced by the zoom factor set using the Set Zoom Factor... option of the Zoom submenu of the Timelines menu.

Zoom Out

When pressed, this icon causes the time interval to be increased by the zoom factor set using the Set Zoom Factor... option of the Zoom submenu of the Timelines menu.

Summarize

When pressed, this icon invokes the Summarize option of the Summary menu which operates on the profile currently selected in the Profile Status List panel. If no profile is currently selected, a summary of all events is executed.

Daemons Toolbar

This toolbar consists of four icons.

Launch

When pressed, this icon launches all daemons currently selected in the Daemons panel.

Resume

When pressed, this icon resumes all daemons currently selected in the Daemons panel.

Pause

When pressed, this icon pauses all daemons currently selected in the Daemons panel.

Halt

When pressed, this icon halts all daemons currently selected in the Daemons panel.

Panels Toolbar

This toolbar consists of seven icons, representing each of the available panel types in NightTrace. When pressed, the icon toggles the visibility of the corresponding panel in the current page.

Pages

The remaining area of the main window is reserved for various tabbed pages which can contain any of the seven panel types available within NightTrace.

Each page has a tab which contains the page title. When clicked or right-clicked, the page is raised to the top and becomes the current page.

Each tab has a context menu which allows you to manipulate the page position and title.

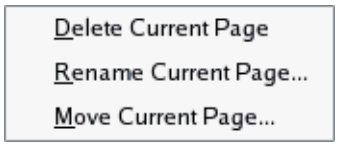


Figure 8-17. Tab Context Menu

Delete Current Page

Mnemonic: D

This option deletes the current page.

Rename Current Page

Mnemonic: R

This option launches a dialog which allows you to rename the current page.

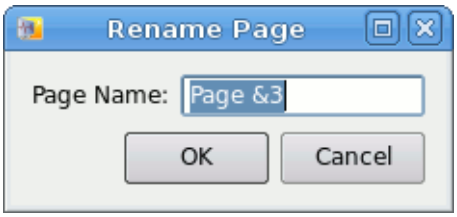


Figure 8-18. Rename Page Dialog

If the page title contains an ampersand character (&), it causes the next character to be underlined, provides a keyboard shortcut for that page, and the ampersand becomes invisible in the title that is shown for the page. In the example above, the keyboard shortcut for this page will be Alt+4 and the displayed title will become Page 4. Activating the shortcut for a page causes it to be raised to the top and it becomes the current page. Care should be taken when choosing shortcuts for pages so they do not conflict with other shortcuts. If you desire to have an ampersand displayed in the actual page title (as opposed to defining a shortcut), use two ampersand characters, back to back in the Rename Page dialog.

Move Current Page

Mnemonic: M

This option launches a dialog which allows you to reposition the current page among other pages. This option will be disabled unless at least two viewing pages exist.

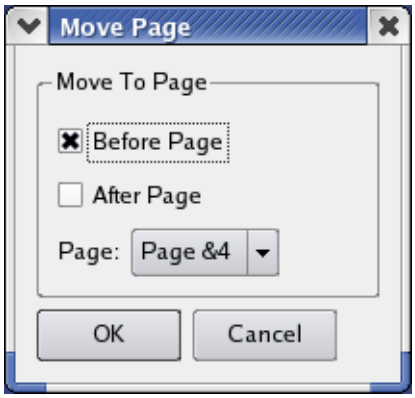


Figure 8-19. Move Page Dialog

Panels

NightTrace provides flexibility in configuring the graphical user interface to suit your needs through the use of resizable and movable panels.

Consider the following page which contains a Profile Definition panel and a Profile Status List panel:

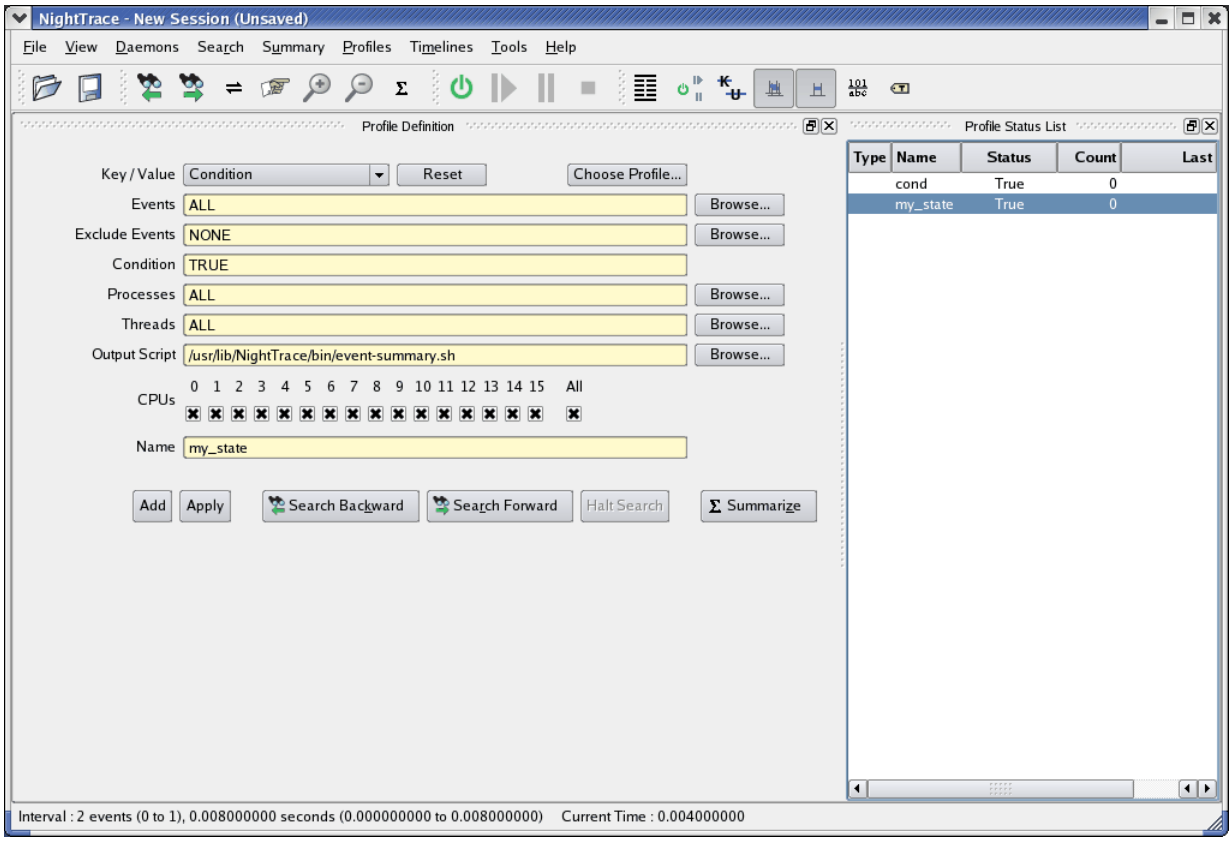


Figure 8-20. Page with Profile Panels

Panels are moved by left-clicking the title bar, dragging them to a new location, and then releasing the mouse button. Depending on the location of the panel when the mouse button is released, the panel will either remain detached or will be inserted into the page again.

To detach the panel from the page without inserting it, click the left-most control box in the upper right-hand corner of the panel.

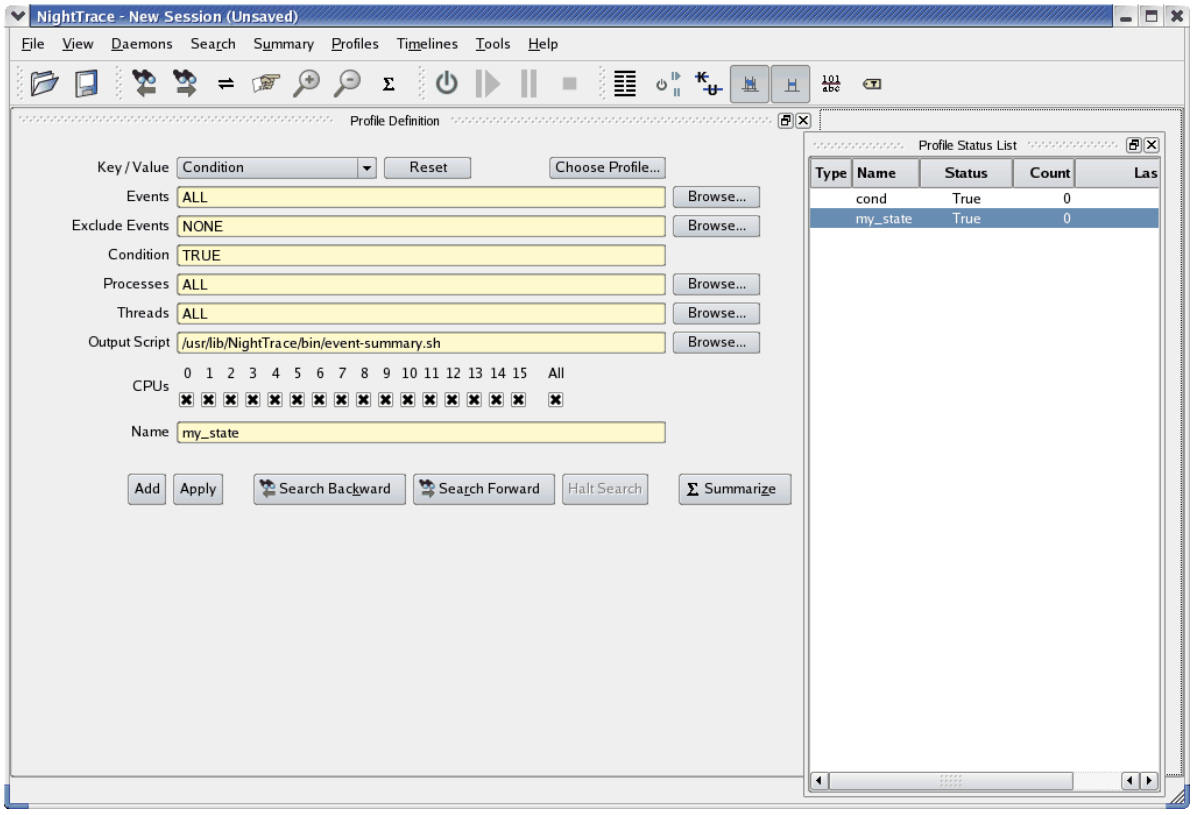


Figure 8-21. Panel Detaches from Page

The Profile Status List panel detaches from the page and becomes free floating. If moved outside the boundaries of the main window and released, the panel will remain detached from the main window. However, even in detached mode, if the main window is iconified, the detached panel will be iconified with it.

To insert a panel into the page at a new location, drag the panel using the left mouse button on its title bar and move it until it approaches a boundary of the page. NightTrace will respond by creating space indicating where the panel will be inserted.

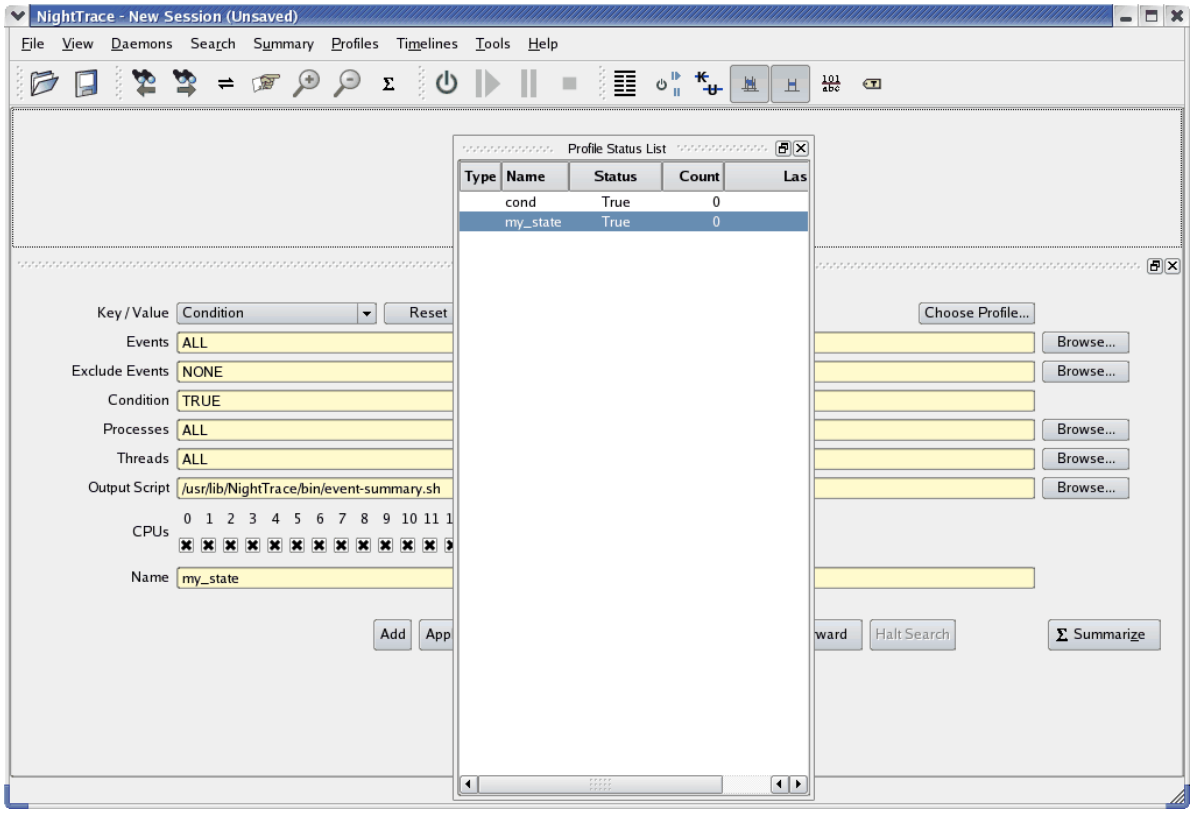


Figure 8-22. Panel Movement in Progress

The figure above shows space being created above the Profile Definition panel as the Profile Status List panel is dragged towards the upper horizontal boundary of the page.

At this point, releasing the mouse button will cause the Profile Status List panel to be inserted into the page, consuming the recently created space.

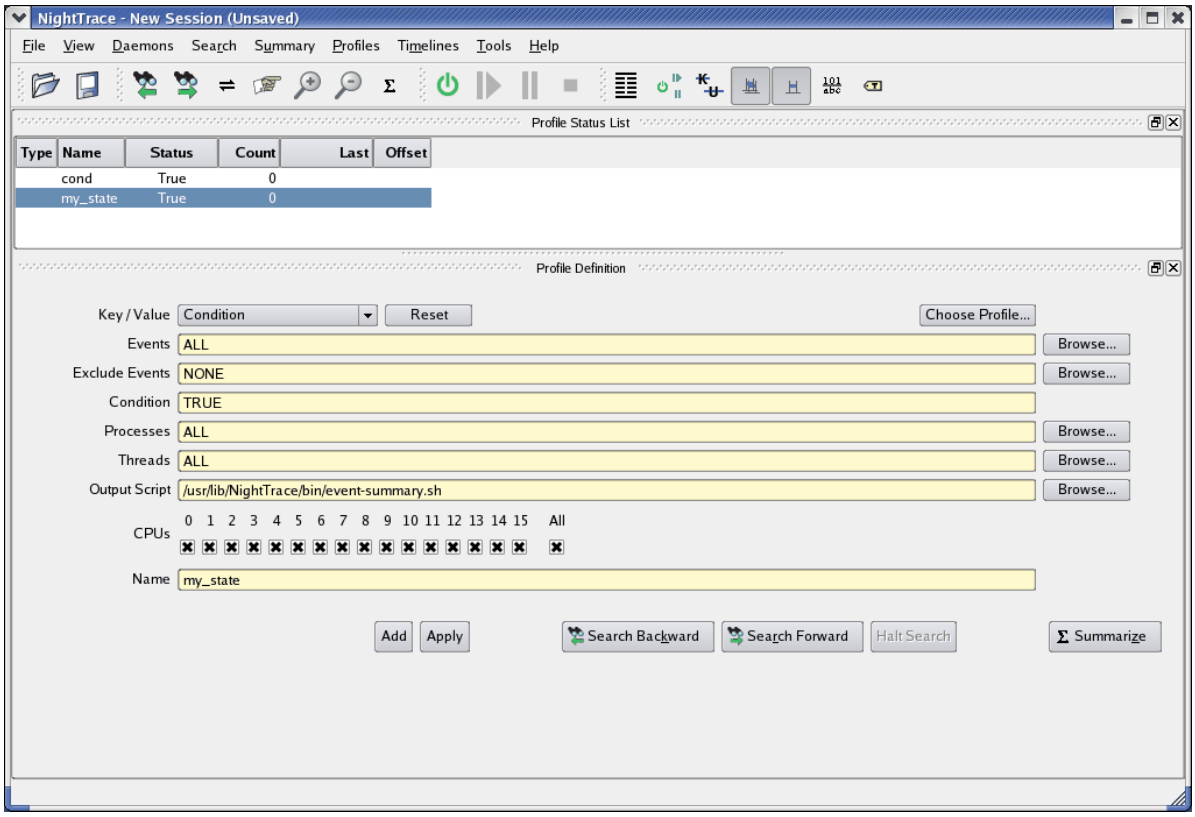


Figure 8-23. Profile Status List Panel on Top of Profile Definition Panel

IMPORTANT

When attempting to move panels inside of a page, if an empty space does not appear where you desire it, try increasing the size of the main window, decreasing the size of the undocked panel, and moving an alternative edge of the undocked panel near where you want to place it.

In the following figure, an Event Descriptions panel has been added to the right-hand side of the Profile Definition and Profile Status List panels.

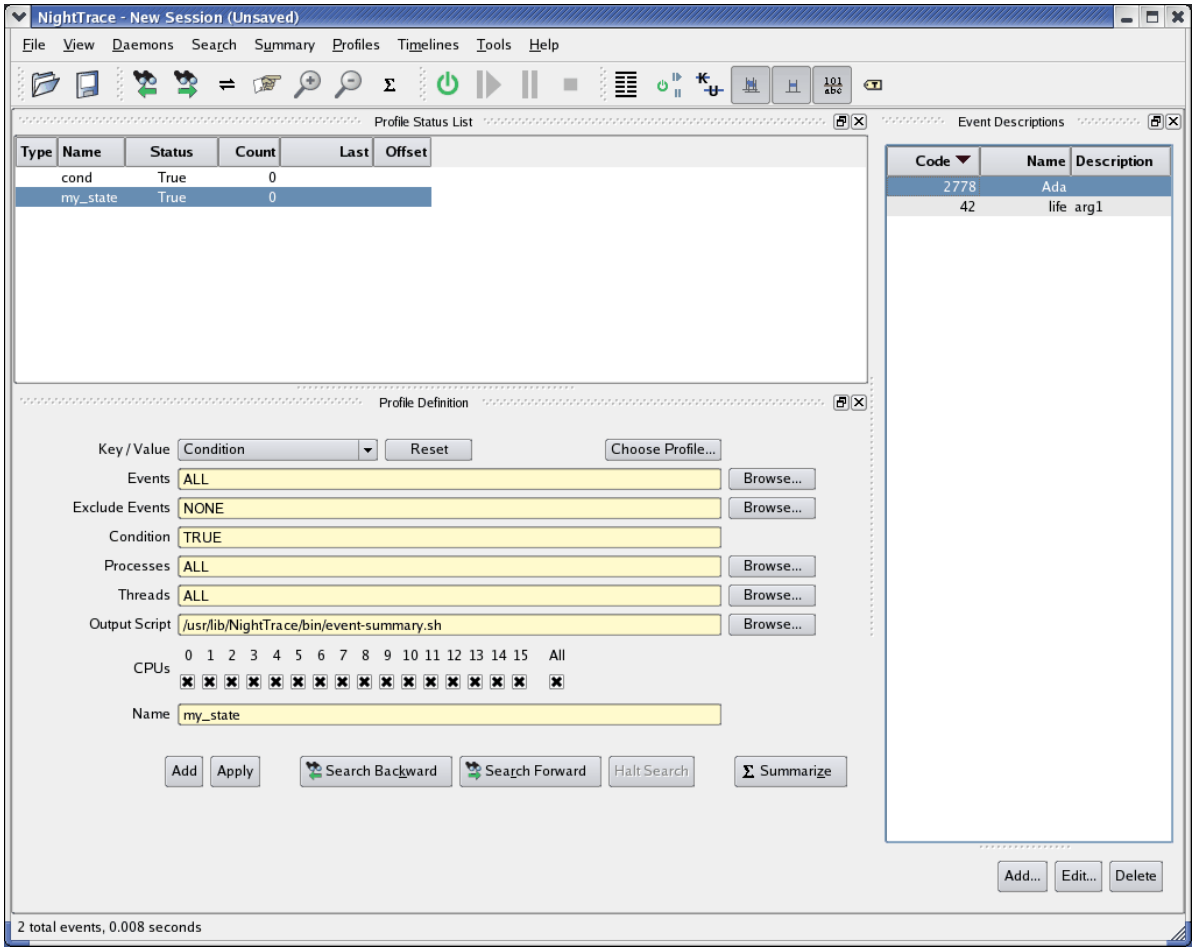


Figure 8-24. Event Descriptions Panel added to Page

Panels can be resized by left-clicking on the separator between the panels and dragging it to the desired size.

Another feature of the graphical user interface is the use of tabbed panels. Tabbed panels allow you to maximize your GUI real estate by placing two or more panels in the same location by stacking them on top of each other. You can then raise a panel to the top by clicking on its tab.

To create a tabbed panel, move a panel to the lower horizontal edge of another panel until a tab appears at the bottom of the panel still connected to the page.

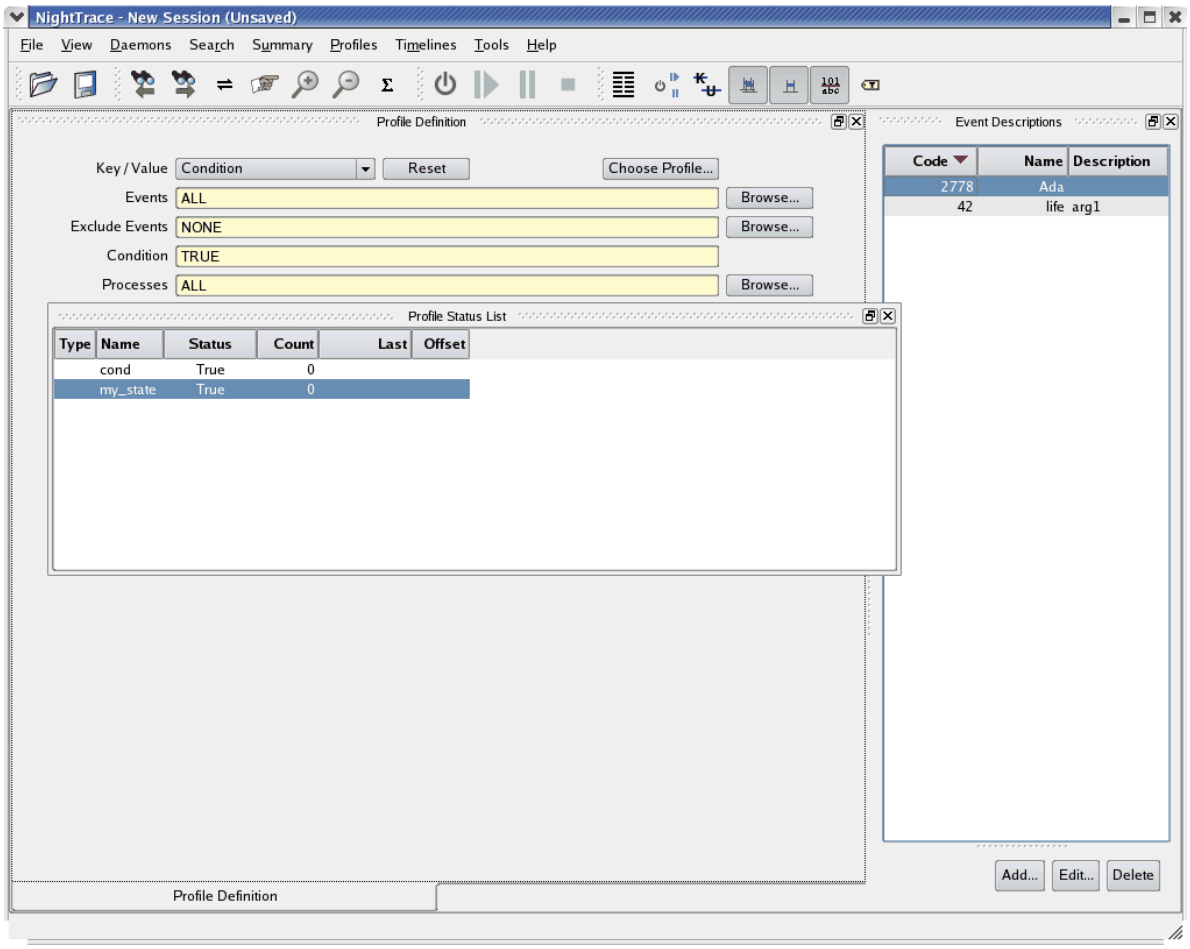
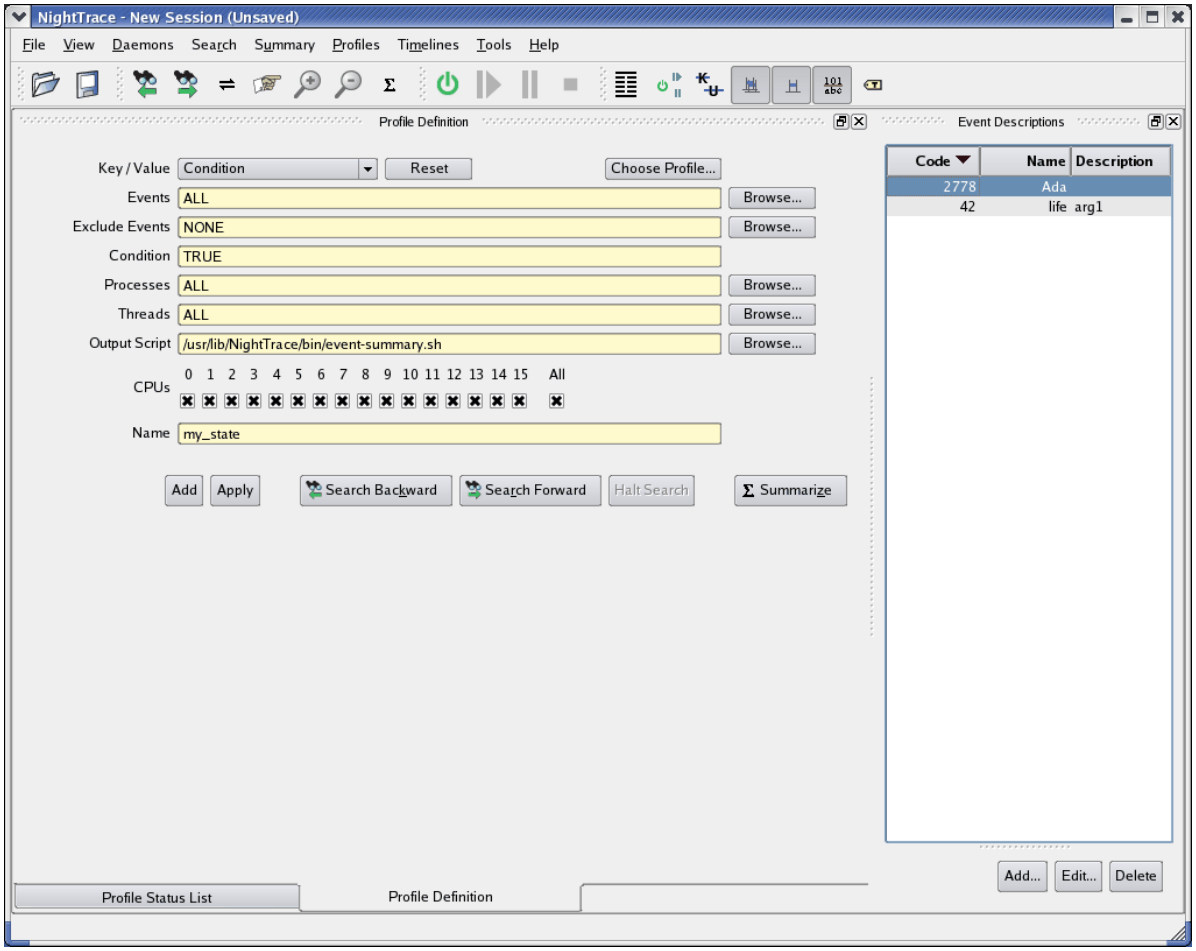


Figure 8-25. Panel in Motion Creating Tab

In the figure above, the Profile Status List panel is being dragged from its original position on top of the Profile Definition panel towards the bottom of the Profile Definition panel. A tab appears on the Profile Definition panel indicating that if the mouse button is released, the Profile Definition and Profile Status List panels will be

tabbed and therefore consume the same area of the page.



IMPORTANT

To move a panel above another panel, move the desired panel to the top boundary of the other panel. If you move a panel to the bottom boundary of another panel, it will become a tabbed panel instead.

The orientation and size of panels within pages is saved as part of a NightTrace session.

Preferences Dialog

The Preferences Dialog is launched via the Preferences... option of the File menu.

Font Preferences

NightTrace uses multiple fonts to present text in the most effective manner throughout the various display areas of the tool.

Variable-width fonts are most commonly used; these fonts most closely resemble how people write or print words.

Fixed-width fonts require that all characters and numbers have the same width (visual footprint). Fixed-width fonts are of benefit when source code is being displayed or manipulated or when columns of numbers are viewed.

NightTrace further divides the use of fonts into the following categories; default, panel, and timeline.

Default fonts are used for text associated with operational description and control, including: menus, buttons, selection devices, labels, tool tips, status bar messages, and generally descriptive verbiage.

Panel fonts are used in NightTrace panels, which display the data of highest importance.

Timeline fonts are used in Event Timelines. Timeline fonts are separated from panel fonts in NightTrace because it may be advantageous to use a small font in timelines where space may be at a premium, especially for kernel display pages on systems with more than a few CPUs.

Fonts are selected by querying font preferences from the following sources until a preference is found:

- Your NightTrace preference
- Your NightStar-wide preference
- The system's NightTrace preference
- The system's NightStar-wide preference
- NightTrace's ultimate default

Figure 8-26. Font Preferences Page

This page is divided into three sections.

Global NightStar Fonts

The Change... button in this area launches the NightStar Global Fonts dialog which allows you to set your Nightstar-wide preferences, your preferences for another specific NightStar tool, or the system's tool or NightStar-wide preferences.

Note:

Setting a NightStar preference for the system typically requires `root` access.

Changes saved in the NightStar Global Fonts dialog are always saved to disk and apply to the current and subsequent NightTrace invocations.

My NightTrace Fonts

This area allows you to set or clear your user's preferences for NightTrace.

Selection of the checkboxes for the individual font categories control whether or not your preferences are to be consulted. Clearing a checkbox effectively removes your user preference for that category. Setting a checkbox allows you to select specific fonts within the category.

Changes to any of the settings in this area, including individual fonts or category checkboxes, are immediately reflected in the Effective NightTrace Fonts area at the bottom of the page so you can see the ultimate effect a change will have.

To change a specific font, ensure that the corresponding category's checkbox is checked and then press the Change… button. This will launch a standard font selection dialog. When you select a font from the dialog and press OK, the name of the font family is displayed to the left of the Change… button and is displayed in the selected font as well.

Effective NightTrace Fonts

This area shows you the effective fonts that will be used based on your user settings and consultation of global settings which aren't shown in the page.

The values in this area immediately change to reflect the effective font whenever any change is made within the page.

Your changes in the My NightTrace Fonts area are applied to the current invocation of NightTrace when you press the OK button. However, your changes are not saved to disk and will not affect subsequent invocations of NightTrace unless you press the Save button.

Separation of Apply and Save operations make it easy to experiment with fonts in the current invocation without affecting long-term usage.

Note:

Changes to font preferences in the NightStar Global Fonts dialog are always saved to disk and apply to the current and subsequent NightTrace invocations; i.e. there is no way to experiment with a global font preference without affecting subsequent NightTrace invocations.

The buttons at the bottom of the page control the application of your changes.

OK

Applies any changes made directly in the Font Preferences page to the current invocation of NightTrace and closes the dialog. The changes will not be saved to disk or affect subsequent NightTrace invocations unless you return to the Preferences… dialog and press Save.

Reset

Discards any changes you have made directly to the Font Preferences page since the dialog was launched and resets the dialog accordingly.

Note:

Changes made in the NightTrace Global Fonts dialog cannot be discarded via the Reset button.

Save

Applies the preferences from the dialog to the current invocation of NightTrace, saves the preferences to disk thereby affecting subsequent NightTrace invocations, and closes the dialog.

Cancel

Cancels any pending changes and closes the dialog.

Help

Opens the help system to display this section.

NightStar Global Fonts Dialog

The NightStar Global Fonts dialog allows you to set your Nightstar-wide preferences, your preferences for another specific NightStar tool, or the system's tool or NightStar-wide preferences.

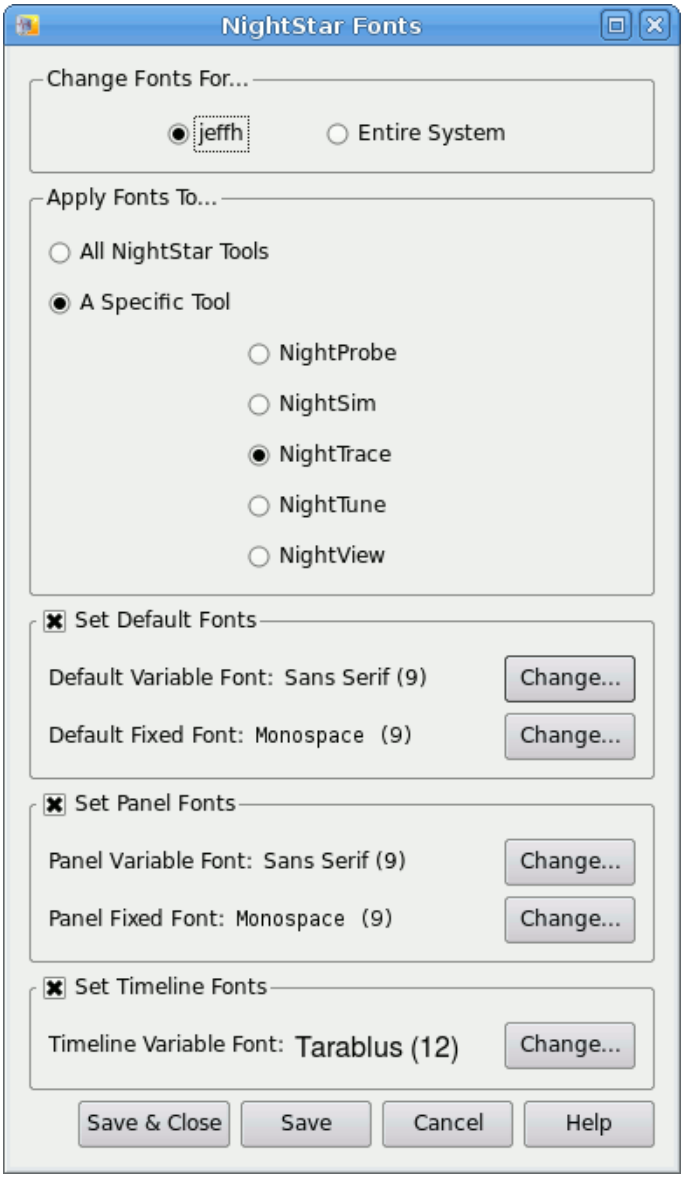


Figure 8-27. NightStar Global Fonts Dialog

Keep in mind that fonts are selected by querying font preferences from the following sources until a preference is found:

- Your NightTrace preference
- Your NightStar-wide preference
- The system's NightTrace preference

- The system's NightStar-wide preference
- NightTrace's ultimate default

This dialog has two control areas which define the scope of font preference application.

Changes Fonts For...

By default, the dialog is set up to apply font preferences to your user account. Select the Entire System button if you wish to set the system's preferences.

Note:

Changing font preference for the system typically requires `root` access.

Apply Fonts To...

This area additionally controls the scope of font preference application. You can change a preference for a specific NightStar tool or change the NightStar-wide preference.

If you wish to change the font for more than one tool from this dialog, but not change the NightStar-wide preference, select the first tool of interest, make your preference change in the areas below, and then press the Save button. Then select the second tool of interest and repeat.

Set Default Fonts
Set Panel Fonts
Set Timeline Fonts

These areas contain the variable and fixed-width font preferences for each of the font categories, identified by the label next to each checkbox.

To remove the preferences in a category, clear its checkbox.

To change a specific font, ensure that the category's checkbox is checked and then press the Change… button. This will launch a standard font selection dialog. When you select a font from the dialog and press OK, the name of the font family is displayed to the left of the Change… button and is displayed in the selected font as well.

The buttons at the bottom of the page control the application of your changes.

Save & Close

Saves any changes made in this dialog to disk, thus affecting subsequent tool invocations, and closes the dialog.

These changes may affect the effective font preferences for the current invocation of NightTrace. When the dialog is closed, the fonts shown in the Effective Night-

Trace Fonts section of the Preferences dialog are updated. If you apply the changes in that dialog, they will take effect in the current invocation of NightTrace.

Save

Applies the preferences from the dialog to the current invocation of NightTrace, saves the preferences to disk thereby affecting subsequent NightTrace invocations.

These changes may affect the effective font preferences for the current invocation of NightTrace. When this dialog is subsequently closed, the fonts shown in the Effective NightTrace Fonts section of the Preferences dialog are updated. If you apply the changes in that dialog, they will take effect in the current invocation of NightTrace.

Cancel

Cancels any unsaved changes and closes the dialog.

Help

Opens the help system to display this section.

9
Daemons Panel

The Daemons panel provides for the creation and control of user and kernel daemons which are used to collect data from user applications and the operating system, respectively.

It is often more convenient to use the Daemons panel to launch and run daemons as opposed to relying solely on the ntraceud and ntracekd command line invocations as described in “Capturing User Events with ntraceud” on page 3-1 and “Capturing Kernel Events with ntracekd” on page 4-1.

Additionally, the Daemons panel aids in locating user applications that are attempting to log trace data yet have no trace daemons currently associated with them. You can also gain control of a previously-executed command line daemon by using the Attach feature of the Daemons panel.

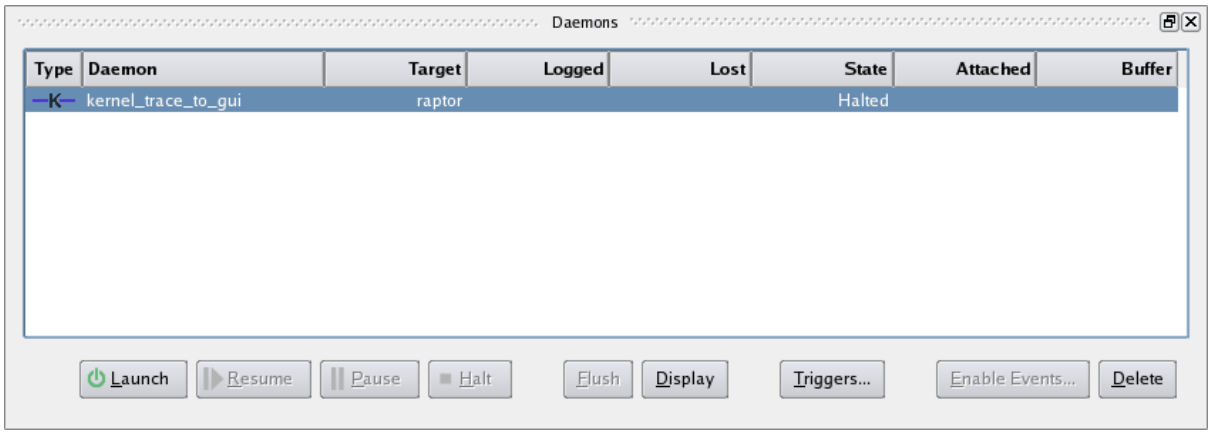


Figure 9-1. Daemons Panel

All daemons defined in the current session are shown as individual rows in the panel.

Using the buttons at the bottom of the panel, you can control the execution of the daemons as well as bring data into NightTrace Timeline panels for immediate viewing.

Context Menu

The panel's context menu provides a super-set of the activities controlled by the buttons at the bottom of the panel, including the ability to create and edit daemon definitions.

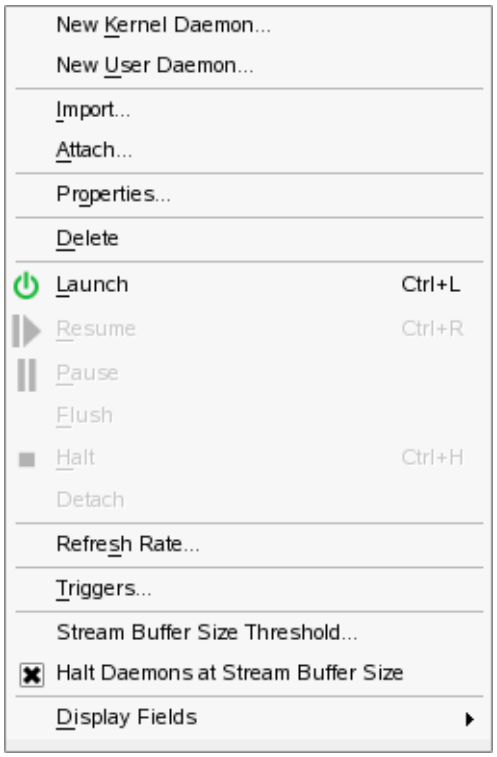


Figure 9-2. Daemons Panel Context Menu

New Kernel Daemon...

Mnemonic: K

Opens the Edit Daemon Definition dialog (see "Edit Daemon Definition" on page 9-8) allowing the user to configure a new kernel daemon definition.

NOTE

Support for kernel tracing is only available on some operating system distributions. See "Kernel Dependencies" on page B-1 for more information.

New User Daemon...

Mnemonic: U

Opens the Edit Daemon Definition dialog (see “Edit Daemon Definition” on page 9-8) allowing the user to configure a new user daemon definition.

Import...

Mnemonic: I

Presents a dialog which lists all user applications on the target system that are attempting to log trace data but that do not currently have user daemons associated with them.

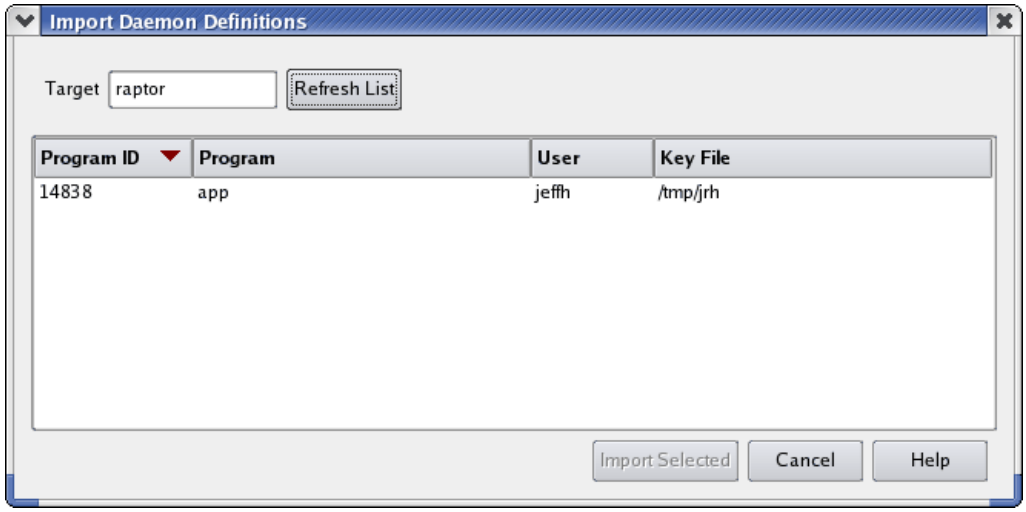


Figure 9-3. Import Daemon Definitions Dialog

Each application that has called `trace_begin()`, but that does not yet have a daemon, is listed in a row in the table.

The table includes the Process ID, Program name, User, and the name of the Key File as passed to `trace_begin()`.

To import any daemon configuration information specified by the user application (the second parameter to `trace_begin()`), click the row of interest and press the Import Selected button.

This causes a daemon definition to be automatically created and the Edit Daemon Definition dialog is launched so you can make any required adjustments, as described in “Edit Daemon Definition” on page 9-8.

Attach...

Mnemonic: A

Allows the user to query any target system for user application trace daemons and displays the results in a dialog.

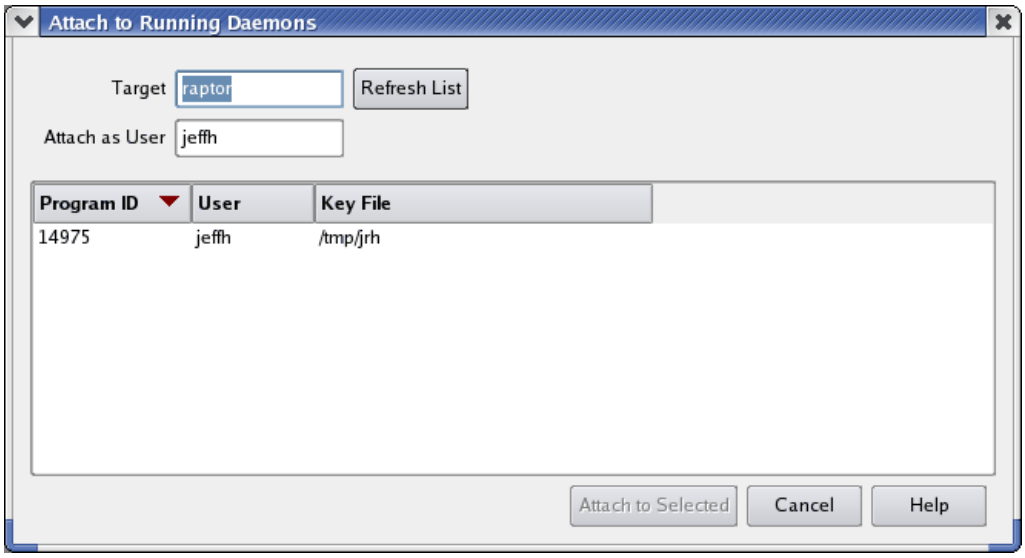


Figure 9-4. Attach to Running Daemons Dialog

The user may then attach to the desired daemon and control it, by selecting a daemon from the list and pressing the Attach to Selected button.

A daemon definition is created for the daemon and it is added to the list of daemons in the panel.

Properties...

Mnemonic: O

Opens the Edit Daemon Definition dialog (see "Edit Daemon Definition" on page 9-8) allowing the user to configure the currently selected daemon.

Delete

Mnemonic: D

Deletes the daemon definition currently selected in the panel.

Launch

Mnemonic: L

Starts execution of the daemon(s) currently selected in the panel.

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.

Resume

Mnemonic: R

Resumes execution of the daemon(s) currently selected in the panel. Once resumed, incoming events are placed into the daemon buffer for subsequent processing by the daemon.

Pause

Mnemonic: P

Pauses the execution of the daemon(s) currently selected in the panel.

NOTE

When a daemon is paused, incoming trace events are discarded without notice.

Flush

Mnemonic: F

Flushes trace events from the buffers associated with the daemon(s) currently selected in the panel to either the NightTrace display buffer or to the output file.

Halt

Mnemonic: H

Stops execution of the daemon(s) currently selected in the panel.

Detach

Relinquishes control of the running daemon(s) currently selected in the panel. Daemons writing to a file will continue to execute and will continue to write events to a file. If the file has no size limit associated with it, it could consume large amounts of disk space.

You cannot detach from a daemon which is streaming events directly to NightTrace.

Refresh Rate...

Mnemonic: S

Provides a dialog which controls the refresh interval of statistics for active daemons as shown in the panel.

Triggers...

Mnemonic: T

This option launches the Edit Triggers dialog.

Triggers allow you to set a condition which is continually evaluated as streaming data is sent to NightTrace. When the condition evaluates to `true`, NightTrace will stop all executing daemons under its control. Daemons with triggers must be streaming data into NightTrace -- daemons writing to files are not eligible for triggers.

See "Triggers" on page 9-15 for more information.

Streaming Memory Usage Control...

Mnemonic: M

This option launches the Streaming Memory Usage Control dialog.

For streaming daemons, the Streaming Memory Usage Control limit defines the maximum amount of memory that NightTrace should use to hold streaming data.

See "Streaming Memory Usage Control" on page 9-18 for more information.

Display Fields

The Display Fields submenu provides checkboxes for each of the column headers that can be displayed in the panel. When checked, the column is present; otherwise the column is hidden.

Control Buttons

At the bottom of the panel there are a series of buttons that operate on daemons that are currently selected in the panel.

Most of the buttons execute obvious actions, as described in detail in the panel's Context Menu. The descriptions below provide a brief summary of those actions as well as detailed descriptions of actions not available in the Context Menu.

Launch

Launches the currently selected daemons. See "Launch" on page 9-4 for more information.

Resume

Resumes the currently selected daemons. See “Resume” on page 9-5 for more information.

Pause

Pauses the currently selected daemons. See “Pause” on page 9-5 for more information.

Halt

Halts the currently selected daemons. See “Halt” on page 9-5 for more information.

Flush

Flushes the internal buffers of the currently selected daemons, forcing the data to be sent to the output device (file or stream attached to NightTrace). See “Flush” on page 9-5 for more information.

Display

This option is equivalent to flush except in the case of a daemon writing to a file. Once such a daemon is stopped, pressing Display will load the contents of the file containing the trace data.

Triggers...

This button launches the Edit Triggers dialog.

Triggers allow you to set a condition which is continually evaluated as streaming data is sent to NightTrace. When the condition evaluates to `true`, NightTrace will stop all executing daemons under its control. Daemons with triggers must be streaming data into NightTrace -- daemons writing to files are not eligible for triggers.

See “Triggers” on page 9-15 for more information.

Enable Events

Launches a dialog which allows you to enable or disable events while the daemon is executing.

Delete

Deletes the currently selected daemons; daemons cannot be deleted until halted.

Edit Daemon Definition

The Edit Daemon Definition dialog allows the user to create and modify the various aspects of a daemon configuration.

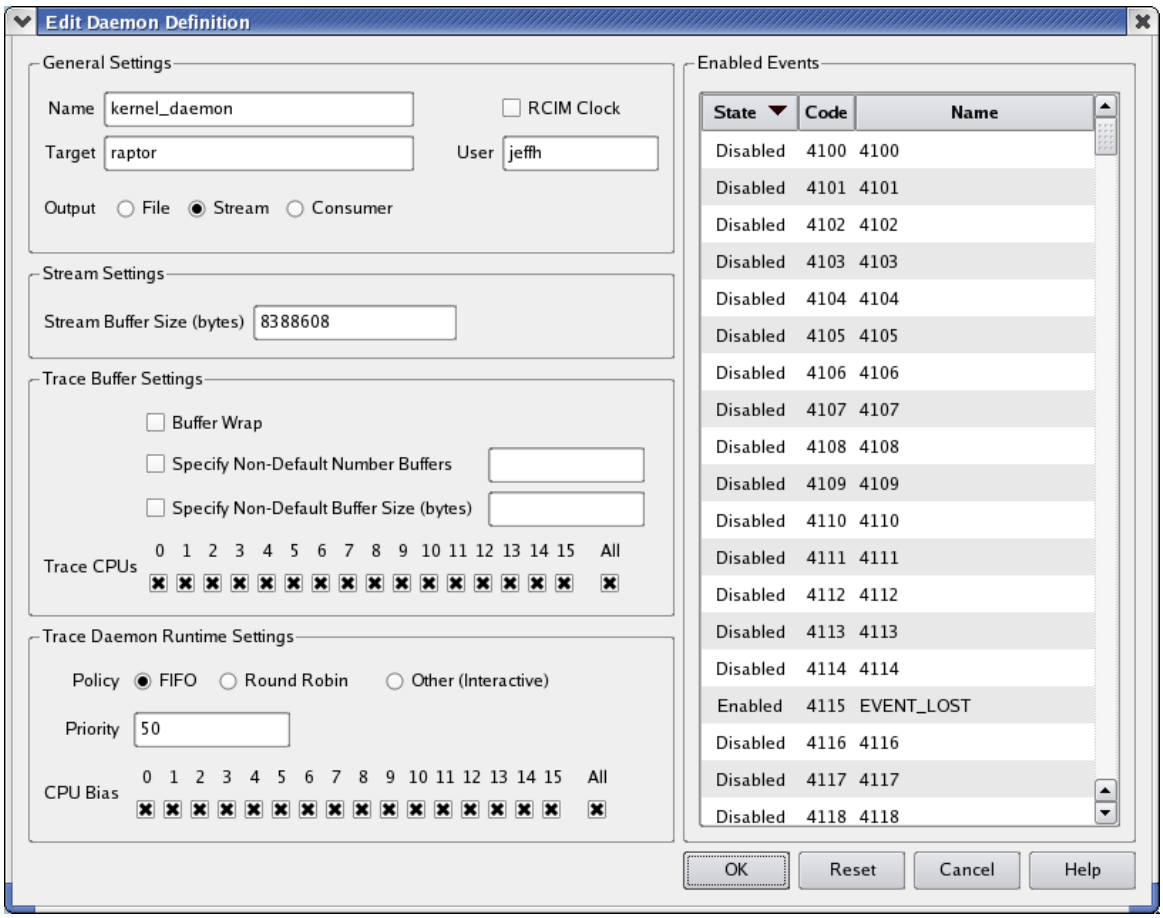


Figure 9-5. Edit Daemon Definition Dialog

The Edit Daemon Definition dialog is divided into a number of areas that contain specific information about the current configuration, including:

- "General Settings" on page 9-9
- Trace File, Stream, and Consumer Output Settings (see "General Settings" on page 9-9)
- "Trace Buffer Settings" on page 9-11
- "Trace Daemon Runtime Settings" on page 9-14
- "Enabled Events" on page 9-15

General Settings

The General area of the dialog contains information such as the name of the daemon configuration, the target system on which the daemon will run, the user name, and the output method.

Name

This field is automatically populated with the name user_daemon or kernel_daemon for each new daemon definition. A..x notation is appended when required, starting at 1, in order to keep the daemon names unique within a NightTrace session.

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

The system on which this trace daemon will run.

RCIM Clock

When checked, the RCIM tick clock will be used to timestamp data. By default, the system's architecture clock is used as a timing source. Use of the RCIM tick clock is advantageous when multiple systems are being traced at the same time and their RCIM clocks are synchronized through an RCIM cable.

User

The name of the user on the specified target system responsible for running this daemon.

Output

These radio buttons define the output method.

File

When selected, all trace data is written directly to a disk file. You cannot analyze the data until the daemon has stopped collecting data and you load it into NightTrace using the Display button in the Daemons panel or the Open Files… option of the File menu in the main window.

Use of the File method requires you to enter information in the Trace File Settings group area which appears immediately below the General Settings area when this method is selected. For kernel daemons, this can be any filename. For user daemons, this must be the pathname the user application specified to the `trace_begin, Trace.begin()` call to initiate tracing.

If you check the File Wrap checkbox, the file size will be limited by the value in the Size Limit (bytes) field. When the limit is reached, the oldest trace data is overwritten with newer trace data.

Stream

When selected, all trace data is streamed directly into the current NightTrace session for immediate analysis. You can analyze trace data as it is collected or save it to a file for subsequent analysis.

You can adjust the Stream Buffer Size (byte) value in the Stream Settings group area which appears immediately below the General Settings group area when this mode is selected. You may wish to increase the size of the internal buffer NightTrace uses to pass data between the daemon and the analysis modules of NightTrace. If this buffer is too small, NightTrace iteratively pauses and resumes the daemon to catch up with processing (in which case you will see P and R markers in timeline rulers indicating the Pause and Resume operations). Normally, the default value is sufficient for most data rates.

This buffer is only used during the transfer of data blocks between the daemon and the analysis modules. It is unrelated to the Streaming Memory Usage Control limit, which sets a boundary for the amount of memory used to hold all trace data for all active streams. See "Streaming Memory Usage Control" on page 9-18 for more information.

Consumer

When selected, all trace data is streamed directly into a user application of your choice. It is assumed that the user application is written using the "NightTrace Analysis Application Programming Interface" on page 18-1.

You must specify the command that launches your application in the Consumer Application field which appears in the Consumer Application Settings group area immediately below the General Settings area when this mode is selected. You may specify arguments in the field as well.

When launched, the `stdin` file descriptor associated with your program is associated with the stream of trace data being generated by the daemon.

Key File

This is required for user daemons. This field does not appear for kernel daemons and it is also hidden for user daemons that specify File output, in which case the filename is specified in the Trace File field as described under File above.

This must be the pathname the user application specified to the `trace_begin, Trace.begin()` call to initiate tracing.

Trace Buffer Settings

The contents of the Trace Buffer Settings area differ depending on whether the daemon is a user or kernel daemon.

User Daemons

Buffer Wrap

When checked, events remain in memory and are not written to the output device until an explicit flush operation is executed. When all buffers are full, the oldest trace events are overwritten with new trace events.

Bufferwrap can be extremely useful in the following situations:

- When an event of interest occurs very infrequently and the trace data of interest is that only leading up to the event.
- When even the activity of writing events from memory to the output device can adversely affect system or application conditions.
- When the trace data rate is so intense that capturing all events overloads the network or NightTrace. Using bufferwrap and examining snapshots using the Flush button can still be useful in these situations.

Default Page Policy

When checked, the default page-locking policy is in effect. The default policy is to leave pages in their default state (which would normally be unlocked unless the user application has taken some action outside of the NightTrace API, such as **`mlock(2)`**).

Lock Critical Pages

When checked, pages in use by the NightTrace API, as well as the shared memory pages associated with daemon buffers and control structures, will be locked in memory.

NOTE

Locking pages requires the user application to run as root or to have privileged capabilities. See **`pam_capability(3)`** for more information on granting privileged access to non-root users.

Inherit Settings

When checked, the daemon will defer to any configuration settings the user application may have specified on the `trace_begin, Trace.begin()` call, if the user application has already started.

When unchecked and the user application has already started, any critical configuration mismatches (e.g. use of an alternative clock, ability to lock pages, etc.) will cause the daemon invocation to fail with an appropriate diagnostic.

Number Buffers

This setting controls the number of shared memory buffers in use between the user application and the daemon. This number, combined with the setting for Buffer Size, defines the total number of raw events that can be held in memory. In default operating mode (i.e. not buffer-wrap), when a single buffer fills, the user application automatically informs the NightTrace daemon and the daemon wakes up and copies the buffer to the output device.

Reducing the number of buffers reduces the number of wakeup events the user application needs to make to the daemon (although these are very short and efficient). However, reducing the number of buffers to a value less than 8 can cause loss of data when trace data rates are high.

The value specified is automatically rounded up to a power of two if it is not already a power of two.

A *raw event* is the amount of storage required to hold an event without arguments. Events with arguments require two or more raw events to hold their data.

Buffer Size

This setting controls the number of raw events that an individual buffer can hold. This setting, combined with the setting for the number of buffers, defines the total number of raw events that can be held in memory.

Increasing the Buffer Size setting is recommended if you have high trace data rates or are losing trace events.

A *raw event* is the amount of storage required to hold an event without arguments. Events with arguments require two or more raw events to hold their data.

Shared Mem Perms

This area allows you to set the permissions to be applied on the shared memory buffer which is used to hold events logged by the user application before they are written to the output device by the user daemon.

Kernel Daemons

Buffer Wrap

When checked, events remain in memory and are not written to the output device until an explicit flush operation is executed. When all buffers are full, the oldest trace events are overwritten with new trace events.

Bufferwrap can be extremely useful in the following situations:

- When an event of interest occurs very infrequently and the trace data of interest is that only leading up to the event.
- When even the activity of writing events from memory to the output device can adversely affect system or application conditions.
- When the trace data rate is so intense that capturing all events overloads the network or NightTrace. Using bufferwrap and examining snapshots using the Flush button can still be useful in these situations.

Specify Non-Default Number Buffers

This setting controls the number of kernel memory buffers in use between the kernel and the daemon. This number, combined with the setting for Specify Non-Default Buffer Size, defines the total number of bytes that can be held in memory. In default operating mode (i.e. not buffer-wrap), when a single buffer fills, the kernel automatically informs the NightTrace daemon and the daemon wakes up and copies the buffer to the output device.

Reducing the number of buffers to a value less than 8 can cause loss of data when trace data rates are high.

The value specified is automatically rounded up to a power of two if it is not already a power of two.

Specify Non-Default Buffer Size

This setting controls the number of bytes that an individual buffer can hold. This setting, combined with the setting for the number of buffers, defines the total number of bytes that can be held in memory.

Increasing the setting is recommended if you have high trace data rates or are losing trace events.

Trace CPUs

These checkboxes specify which CPUs should be traced. Normally, it is best to trace all CPUs in the kernel.

Specifying just a single CPU or a small set of CPUs may be helpful in situations where user applications of interest are bound to otherwise shielded CPUs.

NOTE

Support for kernel tracing is only available on some operating system distributions. See “Kernel Dependencies” on page B-1 for more information.

Trace Daemon Runtime Settings

The Trace Daemon Runtime Settings area allows the user to specify the scheduling policy, CPU bias, and memory binding policies for the daemon.

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` (Other). Each of these scheduling policies is associated with one of the System V scheduler classes.

FIFO

The FIFO (first–in–first–out) policy (`SCHED_FIFO`) is associated with the fixed-priority class in which critical processes can run in predetermined 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 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.

Other (Interactive)

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 good throughput to CPU-bound processes.

Priority

The Priority is relative to the selected Scheduling Policy and the range of allowable values is dependent on the operating system.

On most Linux 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

Selection of a specific CPU or set of CPUs can be advantageous to prevent event loss and reduce daemon intrusion on the rest of the system.

All CPUs

Selects all CPUs on the target system.

Enabled Events

The Enabled Events area allows you to specify which trace event types will be handled by the daemon.

You may also change this list dynamically while the daemon is executing by pressing the Enable Events button in the panel.

User Tracing

By default, all user trace events are enabled.

Kernel Tracing

For kernel trace daemons, the default set of enabled events is highly recommended. You may wish to enable additional events that you may have added to the kernel, a kernel module, or through a kernel event logged through an **`ioctl(2)`** call. See “Additional Kernel Events” on page 17-7 for more information about adding kernel events.

You should not disable kernel events that are enabled by default unless you are an expert in kernel tracing, as it may have an adverse affect on the default kernel display pages generated by NightTrace.

NOTE

Support for kernel tracing is only available on some operating system distributions. See “Kernel Dependencies” on page B-1 for more information.

Triggers

Triggers are conditions that are evaluated as NightTrace analyzes trace events from streaming daemons. (A streaming daemon is one that sends trace data directly to the

`ntrace` tool for immediate processing, as opposed to a daemon that sends such data to a file for subsequent processing).

When a trigger condition is evaluated to true, all streaming daemons are automatically halted and the Current Timeline is set to the event which caused the trigger.

Triggers are useful when you are trying to capture data associated with an event that may occur very rarely.

You may need to capture user and kernel data over a long period of time before the event actually occurs.

With the trigger capability, you can set your conditions, launch your daemons, and then walk away from NightTrace and let it run and capture data until triggered.

When capturing kernel data, or even user data, huge amounts of data may be collected over a fairly short period of time. NightTrace limits the amount of memory it will use to hold streaming trace data via a user-specified setting.

When the memory limit is reached, NightTrace will either halt current daemons or discard the oldest trace events in order to stay under the specified memory limit.

Edit Triggers Dialog

The Edit Triggers dialog is activated by the Triggers… option in the Daemons menu and the Daemons Panel context menu, as well as by the Triggers… button in the Daemons panel.

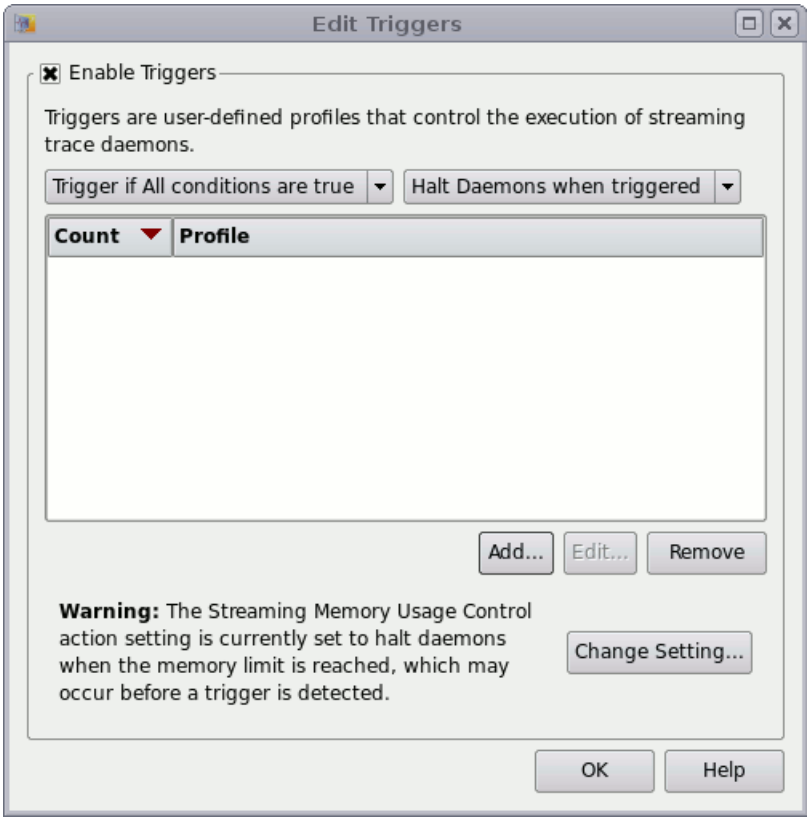


Figure 9-6. Edit Triggers Dialog

Trigger conditions are specified using NightTrace profiles (see “Profiles Panels” on page 13-1).

Enable Triggers

When checked, triggers are enabled, and NightTrace will continually process streaming trace data to evaluate the trigger conditions.

Trigger if All/Any conditions are true

This option list indicates whether All conditions must be true before daemons will be halted, or if only one (Any) of the conditions must be true.

Add

Pressing the Add button launches a dialog which allows you to select an existing profile and optionally apply a count criteria to it.

Figure 9-7. Add Triggers Entry Dialog

Edit

Pressing the Edit button launches a dialog which allows you to change the selected profile and count criteria.

Remove

Pressing the Remove button removes all selected profiles from the list.

Change Setting...

The warning text and the Change Setting… button will only appear if the current Streaming Memory Usage Control action is set to stop daemons when the memory limit for streaming events is exceeded.

This may cause your daemons to be halted before the trigger condition of interest has occurred. Typically, when using triggers, you will want to change the Streaming Memory Usage Control action such that the oldest trace events are discarded when the memory limit is exceeded.

Pressing the Change Setting… button launches a dialog which allows you to change the memory limit and set the associated action. See "Streaming Memory Usage Control Dialog" on page 9-19

Streaming Memory Usage Control

When daemons stream trace data directly to NightTrace for immediate analysis, the trace events are kept in memory. You can set the limit for the total amount of memory to be

used to hold streamed events. You can also instruct NightTrace as to which action to take when the limit is reached; halt streaming daemons or discard the oldest trace events.

Streaming Memory Usage Control Dialog

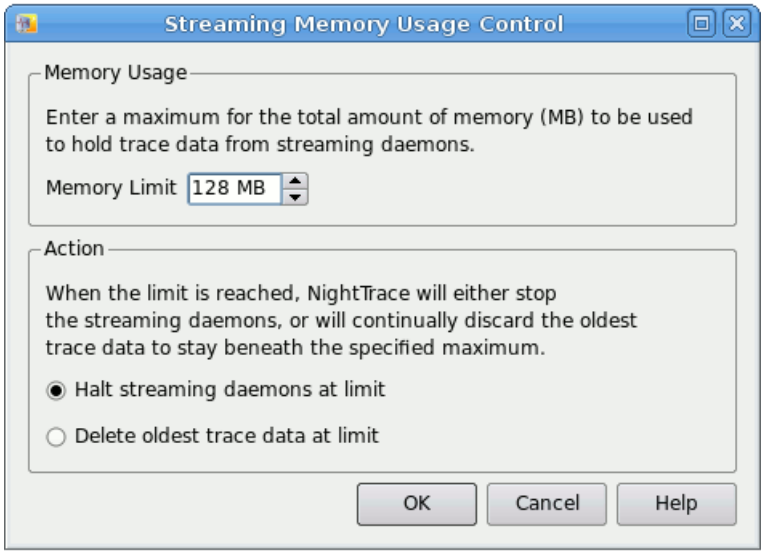


Figure 9-8. Streaming Memory Usage Control Dialog

Memory Usage

Enter the maximum amount of memory, in megabytes, that NightTrace should use to hold streaming trace data. When the limit is reached, the Action criteria defines what action NightTrace will take.

Action

Select the desired action for NightTrace to take when the amount of memory required to hold streaming trace data exceeds the maximum limit set above.

If you have enabled Triggers (See “Triggers” on page 9-15), then you will most likely want to set the action to Delete oldest trace data at limit. Otherwise, the daemons may be shut down before your triggering condition actually occurs.

10
Trace Segments Panel

The Trace Segments panel describes individual trace data segments that are loaded into the current NightTrace session.

Trace Segments Table

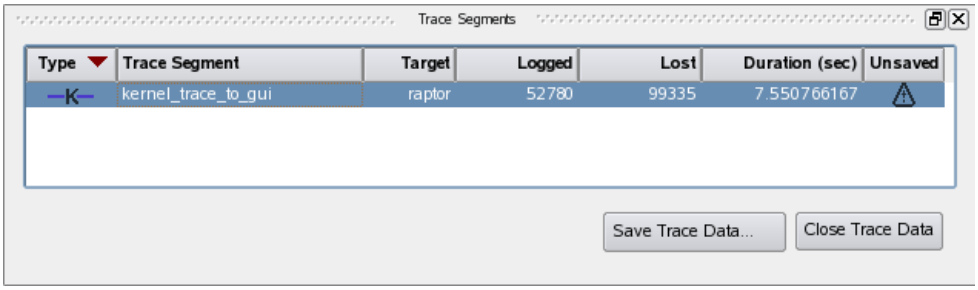


Figure 10-1. Trace Segments Panel

A trace data segment represents data collected from a single user or kernel daemon.

Type

This column provides an icon which indicates whether the daemon is a user daemon or kernel daemon (U or K), and whether it is a streaming daemon (a horizontal line through the letter).

NOTE

Kernel tracing is on support under certain operating system distributions. See “Kernel Dependencies” on page B-1 for more information.

Trace Segment

This column provides the name of the segment which is used merely for identification purposes within a NightTrace session.

Target

This column indicates the target system name where the data was collected.

Logged

This column provides a count of the actual number of events present in the data set. This number almost always differs from the statistics shown in the Daemons panel. The event counts in that panel are raw events. Processed events often consume more than one raw event.

Lost

This column displays a count of the number of raw events that have been lost between the logging agent (kernel or user application) and the daemon.

Event loss can occur for a variety of reasons. See "Preventing Trace Event Loss" on page 6-1 for more information.

When events are lost, an L character appears on trace display Timelines indicating the time at which the loss was recorded.

Duration

This column displays the duration of the data segment.

Unsaved

This column displays an icon indicating the data segment has not yet been saved to disk. This occurs when streaming trace data into NightTrace.

Context Menu

The Trace Segment panel's context menu is shown below:

Figure 10-2. Trace Segment Panel Context Menu

Open Trace File...

This option launches a standard file browser that allows you to select a NightTrace data file to be loaded into the current session.

Save Trace Data...

This option saves all the selected data segments to a NightTrace segment file which can be reloaded in subsequent NightTrace sessions. While the segment file is saved as a single entity, the distinction of the individual data segments is not lost when reloading.

Properties...

This option displays a simple dialog with details of the internal header information embedded in the NightTrace data segment.

It is primary intended for use by NightTrace developers, but it does include generally useful information about the data segment, including the system name, the clock used for timing, and the rate at which the clock ticks.

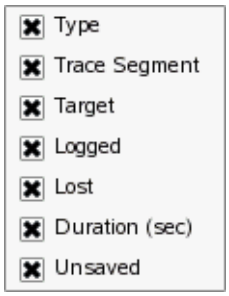
Figure 10-3. Trace Data Segment Properties Description Dialog

Close Trace Data

This option deletes the selected data segments from the current session. All events associated with them are discarded. If the events were streamed into NightTrace and have not yet been saved, a dialog will give you the opportunity to save them before closing them.

Display Fields

This option displays a sub-menu which allows you to customize which columns are visible in the Trace Segments panel.



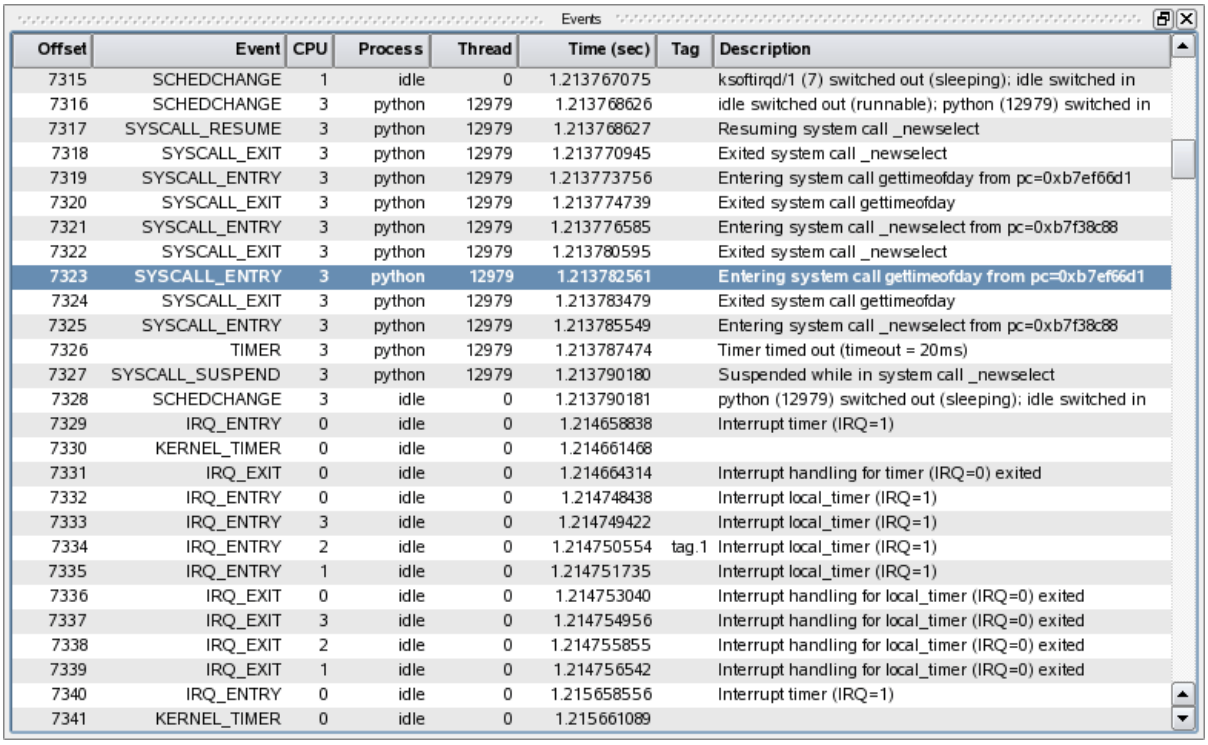
Control Buttons

The buttons at the bottom of the panel provide save and close operations on the selected trace segments, as described in "Save Trace Data..." on page 10-3 and "Close Trace Data" on page 10-4.

11
Events Panel

The Events panel provides a textual table describing all trace events in all trace segments in chronological order.

Textual Event Tables



Offset	Event	CPU	Process	Thread	Time (sec)	Tag	Description
7315	SCHEDCHANGE	1	idle	0	1.213767075		ksoftirqd/1 (7) switched out (sleeping); idle switched in
7316	SCHEDCHANGE	3	python	12979	1.213768626		idle switched out (runnable); python (12979) switched in
7317	SYSCALL_RESUME	3	python	12979	1.213768627		Resuming system call _newselect
7318	SYSCALL_EXIT	3	python	12979	1.213770945		Exited system call _newselect
7319	SYSCALL_ENTRY	3	python	12979	1.213773756		Entering system call gettimeofday from pc=0xb7ef66d1
7320	SYSCALL_EXIT	3	python	12979	1.213774739		Exited system call gettimeofday
7321	SYSCALL_ENTRY	3	python	12979	1.213776585		Entering system call _newselect from pc=0xb7f38c88
7322	SYSCALL_EXIT	3	python	12979	1.213780595		Exited system call _newselect
7323	SYSCALL_ENTRY	3	python	12979	1.213782561		Entering system call gettimeofday from pc=0xb7ef66d1
7324	SYSCALL_EXIT	3	python	12979	1.213783479		Exited system call gettimeofday
7325	SYSCALL_ENTRY	3	python	12979	1.213785549		Entering system call _newselect from pc=0xb7f38c88
7326	TIMER	3	python	12979	1.213787474		Timer timed out (timeout = 20ms)
7327	SYSCALL_SUSPEND	3	python	12979	1.213790180		Suspended while in system call _newselect
7328	SCHEDCHANGE	3	idle	0	1.213790181		python (12979) switched out (sleeping); idle switched in
7329	IRQ_ENTRY	0	idle	0	1.214658838		Interrupt timer (IRQ=1)
7330	KERNEL_TIMER	0	idle	0	1.214661468		
7331	IRQ_EXIT	0	idle	0	1.214664314		Interrupt handling for timer (IRQ=0) exited
7332	IRQ_ENTRY	0	idle	0	1.214748438		Interrupt local_timer (IRQ=1)
7333	IRQ_ENTRY	3	idle	0	1.214749422		Interrupt local_timer (IRQ=1)
7334	IRQ_ENTRY	2	idle	0	1.214750554	tag.1	Interrupt local_timer (IRQ=1)
7335	IRQ_ENTRY	1	idle	0	1.214751735		Interrupt local_timer (IRQ=1)
7336	IRQ_EXIT	0	idle	0	1.214753040		Interrupt handling for local_timer (IRQ=0) exited
7337	IRQ_EXIT	3	idle	0	1.214754956		Interrupt handling for local_timer (IRQ=0) exited
7338	IRQ_EXIT	2	idle	0	1.214755855		Interrupt handling for local_timer (IRQ=0) exited
7339	IRQ_EXIT	1	idle	0	1.214756542		Interrupt handling for local_timer (IRQ=0) exited
7340	IRQ_ENTRY	0	idle	0	1.215658556		Interrupt timer (IRQ=1)
7341	KERNEL_TIMER	0	idle	0	1.215661089		

Figure 11-1. Events Panel

The current timeline is displayed in the panel as the selected event. By selecting a new event in the panel, the current timeline is changed. Thus the Events panel is synchronized with all Timeline panels.

The Events panel table consists of the following columns:

Offset

This column displays the ordinal event offset number within the combined trace data set for the session. The first event in chronological time order has offset zero, the second offset one, and so on.

This is the same value as would be returned by the NightTrace `offset()` function.

Event

This column displays the event ID as a numeric value, or using the corresponding event name, if one exists. Event IDs maybe assigned event names by using the Edit Current Event Description... option of the Event panel context menu, or using the Event Descriptions Panel panel.

CPU

This column displays the CPU where the event was logged for kernel data only. For user events, the CPU information is not available and the value will be displayed as "??".

Process

This column displays the process name that logged the trace event. If a process name is not available, the process ID is used.

Thread

This column displays the thread name or thread ID associated with the trace event. Kernel trace events normally do not have thread names associated with them, unless the user trace data segment is loaded with the kernel trace data and individual threads within the user application were named with a call to `trace_set_thread_name`. See "trace_set_thread_name, Trace.setThread-Name" on page 2-26.

Time

This column displays the time of the event, in seconds, relative to the first event in the combined data set.

Tag

This column displays an event's tag name, if present. Events of interest can be tagged by double-clicking any cell in the row of an event. You can also create a tag by double-clicking an event in a Timeline panel or double-clicking in a ruler in a Timeline panel

Tags allow you to quickly locate events of interest. Tag names are saved as part of a NightTrace session so you can refer to them subsequently. You can annotate a tag with descriptive text using the Tags List Panel or using the context-menu of a tag in a ruler in a Timeline panel (see "Timeline Panels" on page 12-1 for more information).

Description

This column displays an event's description. By default, kernel event descriptions are already associated with all kernel event IDs. For events without descriptions, the values of any arguments are displayed.

You can customize an event's description using the Event Descriptions Panel or by invoking the Edit Current Event Description... option of the Events panel's context menu.

Context Menu

The Events panel context menu is shown below.

Figure 11-2. Events Panel Context Menu

Text Search

Accelerator: Ctrl+T

This option launches the Search Events for Text dialog which allows you to search for specific entries in the Events panel. It does not search for text in Timeline panels.

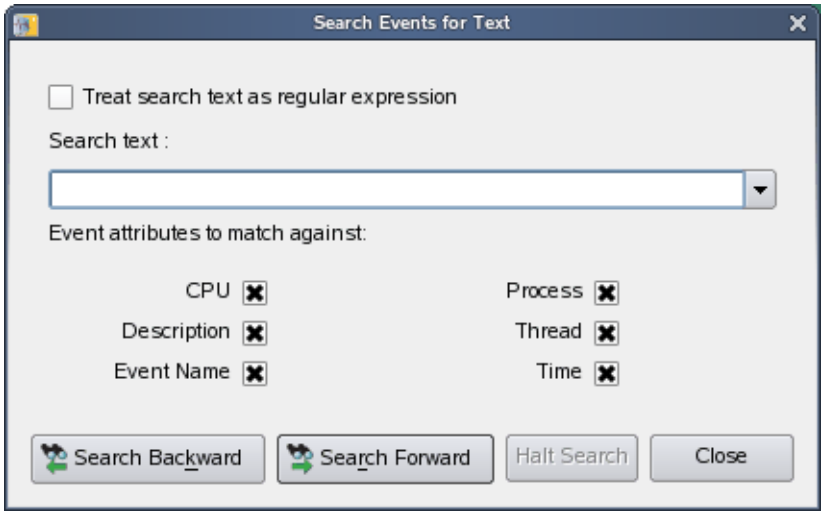


Figure 11-3. Search Events for Text Dialog

Searching occurs for the specified text in cells within the table as controlled by the selected attributes in the dialog.

Treat search text as regular expression

When checked, the text entered in the Search Text field is interpreted as a regular expression as defined by **`regex(3)`**; otherwise, the search is executed for the exact text entered.

Search Text

The text to search for or the regular expression to search for, dependent on the Treat search text as regular expression checkbox.

Event attributes to match against

The search is limited to text associated with table cells corresponding to the attributes checked in this section.

Search Forward

Mnemonic: R
Accelerator: Ctrl+G

Executes a forward search on the previously defined text search. If no such text search has been defined, it searches for the immediately following event.

IMPORTANT

When the focus is in an Events panel, Ctrl+G execute a textual search of that panel. However, when the focus is in a Timeline panel, Ctrl+G executes an event search as defined by the currently selected profile.

Search Backward

Mnemonic: K
Accelerator: Ctrl+B

Executes a backward search on the previously defined text search. If no such text search has been defined, it searches for the immediately preceding event.

IMPORTANT

When the focus is in an Events panel, Ctrl+B execute a textual search of that panel. However, when the focus is in a Timeline panel, Ctrl+B executes an event search as defined by the currently selected profile.

Goto...

This option launches a dialog which allows you to type in an integer event offset value or a floating point number which is interpreted as a time stamp. Pressing OK on the dialog causes the current timeline to move to the specified location.

Distinguish Process Name by PID

This option changes the description of process names to append their process ID. This can be useful when you have multiple processes of interest that have the same simple name.

Edit Current Event Description...

This option launches the Edit Event Description dialog which allows you to define or change the name of an event and its description.

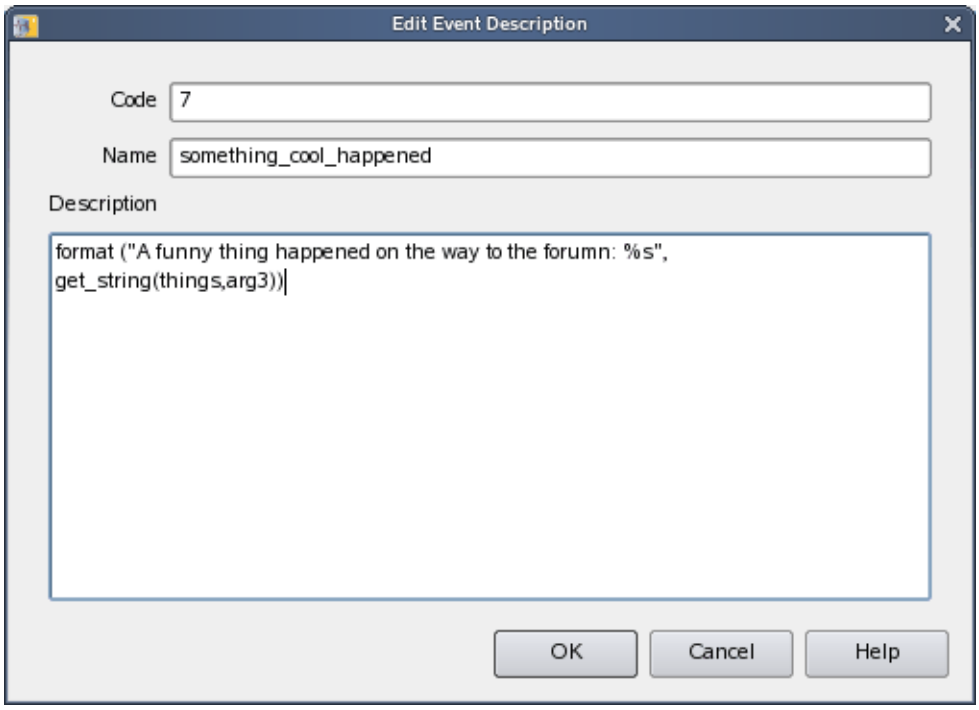


Figure 11-4. Edit Event Description Dialog

Code

This field contains the event ID of interest.

Name

This field defines the textual name that will be displayed in lieu of the event ID.

Description

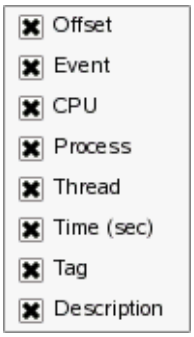
This field allows you to use the NightTrace `format()` function to define a (possibly complex) textual description of the event and its arguments.

Close All Trace Data

This option closes all trace data segments; if some segments have not yet been saved, a dialog gives you the opportunity to cancel the operation.

Display Fields

This option presents the following sub-menu which allows you to select the columns to be displayed in the table:



12
Timeline Panels

A *timeline panel* allows you to analyze trace events both graphically and textually.

Default Timeline

There are two basic types of default timelines; user timelines and kernel timelines. Both operate in essentially the same manner, but a kernel timeline is automatically tailored to aid in viewing kernel events.

The figure below is an example of a default user timeline (see “Kernel Timelines” on page 17-12 for a kernel timeline example).

Figure 12-1. Default User Timeline

A default user timeline consists of the following areas.

- Current Timeline Indicator
- Global Ruler

- Interval Ruler
- Event Graphs
- Event Description Area

The timeline is laid out horizontally and displays trace events as they occurred over time. Events to the left occurred chronologically before events to the right.

The timeline display is interactive. It reacts to zoom, search, and positioning operations.

Current Timeline Indicator

The Current Timeline Indicator is a vertical dashed line which spans much of the vertical area of a timeline. It represents the current time and is synchronized with all other panels throughout the current NightTrace session.

Clicking anywhere within a ruler or event graph in a timeline moves the current timeline. It also responds to search operations throughout NightTrace.

Global Ruler

The Global Ruler is the bottom-most ruler in the timeline.

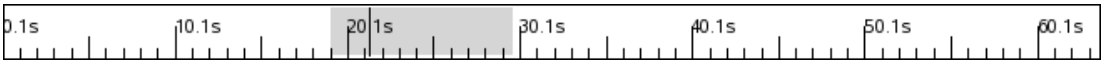


Figure 12-2. Global Ruler

This ruler is the basic mechanism used for moving throughout the entire trace data set with the mouse.

The ruler is annotated with hash marks with time values in units of seconds. It represents the entire data set, not just the data that is currently viewed (also known as the current interval).

The portion of the ruler that has a gray background represents the section of the entire data set that comprises the current interval -- that is, the events that are currently visible in the timeline. Inside the gray area is a single vertical black line which extends through the entire height of the ruler. It represents the location of the current timeline within the current interval.

NOTE

If the current interval is sufficiently small, the width of the gray area may be indistinguishable from the vertical black line within it.

To change the current interval, simply click anywhere in the global ruler. Hence, to look at data near the end of the data set, click very near the end in the global ruler.

See “Keyboard Traversal” on page 12-7 for valuable information on how to use the keyboard to traverse within the current interval and throughout the entire data set.

Interval Ruler

The Interval Ruler is the ruler just above the Global Ruler.

Figure 12-3. Interval Ruler

The Internal Ruler represents the current interval. It is annotated with hash marks with time values in seconds.

Clicking anywhere in the ruler changes the current timeline to that location.

See “Keyboard Traversal” on page 12-7 for valuable information on how to use the keyboard to traverse within the current interval and throughout the entire data set.

The interval ruler can also contain additional objects, as described below.

Tags 

A tag icon is displayed on the ruler for any tag associated with that time. Tags are convenient ways of marking events of interest. They can be annotated with user comments and are saved across NightTrace sessions.

To create a tag using the timeline, double-click a location in the Interval Ruler. You can then annotate the tag by right-clicking on its icon and selecting Annotate… from the context menu.

See “Tags List Panel” on page 15-1 for more information.

Daemon Paused 

This icon is displayed when a daemon is Paused. Events are no longer collected until the daemon is resumed.

NOTE

If the incoming data rate in streaming mode exceeds NightTrace's ability to pass data from the daemon to the display buffer, NightTrace automatically pauses and resumes the daemon in order to catch up. You can increase the Stream Buffer Size using the Daemons Definition dialog to avoid this.

Daemon Resumed 

This icon is displayed when a daemon is Resumed.

Lost Data 

This icon is displayed when event loss is detected. It is associated with an NT_LOST_DATA event, which is not normally displayed in event graphs; however, you can explicitly search for this event. The first argument to the event contains the number of events that were lost.

When event loss occurs, all states currently active in state graphs are terminated and all knowledge of which processes were executing on which CPUs are lost until the next context switch event occurs on each CPU, respectively. (See "Primary Kernel Trace Events" on page 17-1 for more information on kernel event and state graphs).

Event loss can occur for a variety of reasons. See "Preventing Trace Event Loss" on page 6-1.

Time Warp 

This icon is displayed when an internal inconsistency is detected within timestamps. This is most often indicative of a system problem or an internal operating system issue. This is essentially an internal operating system or hardware error, but instead of throwing all data away, NightTrace marks the data set and continues as best it can.

Event Graphs

An Event Graph is a rectangular area within a timeline which contains vertical lines representing events of interest.

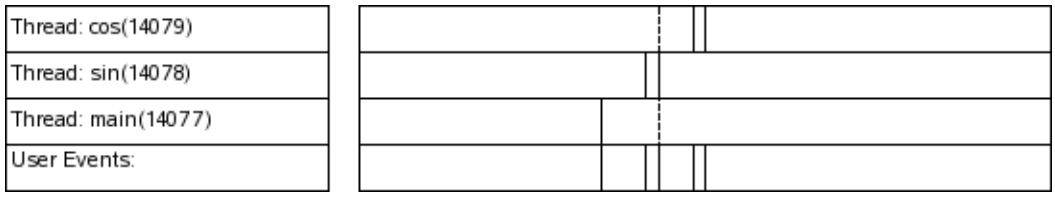


Figure 12-4. Event Graph with Labels

The graphic above shows data boxes on the left hand side which react to changes in the current timeline.

The event graphs on the right display a vertical line when at least one event occurs at that location. Zooming in may provide more detail and the single vertical line may expand to indicate individual events.

Event graphs can be tailored to display events meeting only certain criteria. See “Creating Timeline Objects” on page 12-8 for information on creating and modifying event graphs.

In a default user timeline, an event graph is created for each thread that has logged trace events (if the application has been linked with the thread-aware version of the NightTrace Logging API library). (See “Threads and Logging” on page 2-33 for more information). Each of these graphs only displays events logged by their respective thread. The bottom-most event graph in a user timeline represents all user events -- those logged by any thread, registered or not.

A textual description of the closest event immediately preceding the current timeline is displayed in right-hand portion of the Event Description Area at the bottom of the panel.

As you hover the mouse cursor over any event in the event graphs, a textual description of the event under the mouse cursor is displayed in the left-hand portion of the Event Description Area at the bottom of the panel.

Event Description Area

The Event Description Area provides a textual description of the events.

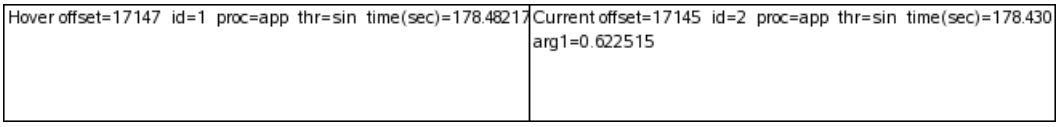


Figure 12-5. Event Description Area

The area consists of two rectangular text areas.

Hover Event Description

The area on the left-hand side describes the event immediately under the mouse cursor. As you move the mouse throughout the timeline and hover over an event, this area updates. If multiple events reside under the mouse cursor, the hover area indicates this. You must zoom in to obtain individual event information in such cases.

The detailed textual description in this area includes the timespan between the hover event and the current timeline.

TIP

To determine the amount of time between two events within the current interval, set the current timeline on one event and then hover the mouse cursor over the second event of interest.

To determine the amount of time between two events which are not both visible in the current timeline, either zoom out so both events are visible or tag each event and use the Tags List Panel to examine the timespans.

Current Event Description

The area on the right-hand side describes the current event. The *current event* is the event immediately at the current timeline or the event most closely preceding it in time.

Event descriptions are provided by default by NightTrace. You can control how events are described by providing customized event descriptions using the Event Descriptions Panel.

Keyboard Traversal

Timelines are designed to be efficiently traversed through keyboard shortcuts when the window focus is in a timeline.

The following table describes keyboard traversal.

Table 12-1. Timeline Keyboard Traversal

Key Sequence	Action
RightArrow	Moves the current timeline to the next event in time
LeftArrow	Moves the current timeline to the previous event in time
UpArrow	Zooms Out
DownArrow	Zooms In
Alt+UpArrow	Zooms all the way out
Alt+DownArrow	Zooms all the way in
Alt+LeftArrow	Goes to the first event in the data set
Alt+RightArrow	Goes to the last event in the data set
Ctrl+RightArrow	Shifts the current interval to the right
Ctrl+LeftArrow	Shifts the current interval to the left
Ctrl+F	Displays the Profile Definition Panel to allow you to define or select a search criteria
Ctrl+G	Executes a forward search using the currently selected profile in the Profile Status List Panel. If no profile is selected, it searches for the next event.
Ctrl+B	Executes a backward search using the currently selected profile in the Profile Status List Panel. If no profile is selected, it searches for the previous event.
Ctrl+I	Launches the Goto dialog which allows you to enter times or offsets that control which events are displayed in the interval.
Alt+G	Identical to Ctrl+G except that the search is constrained by the bounds of the current interval.
Alt+B	Identical to Ctrl+B except that the search is constrained by the bounds of the current interval.
Alt+V	Toggles between the current timeline and the last location of the current timeline. This is especially useful for returning to the previous location after executing a search.

In addition to keyboard shortcuts, moving the mouse wheel back and forth causes the timeline to zoom in and out.

Creating Timeline Objects

Timeline objects can be created or modified by entering Edit mode using the context menu of a Timeline panel.

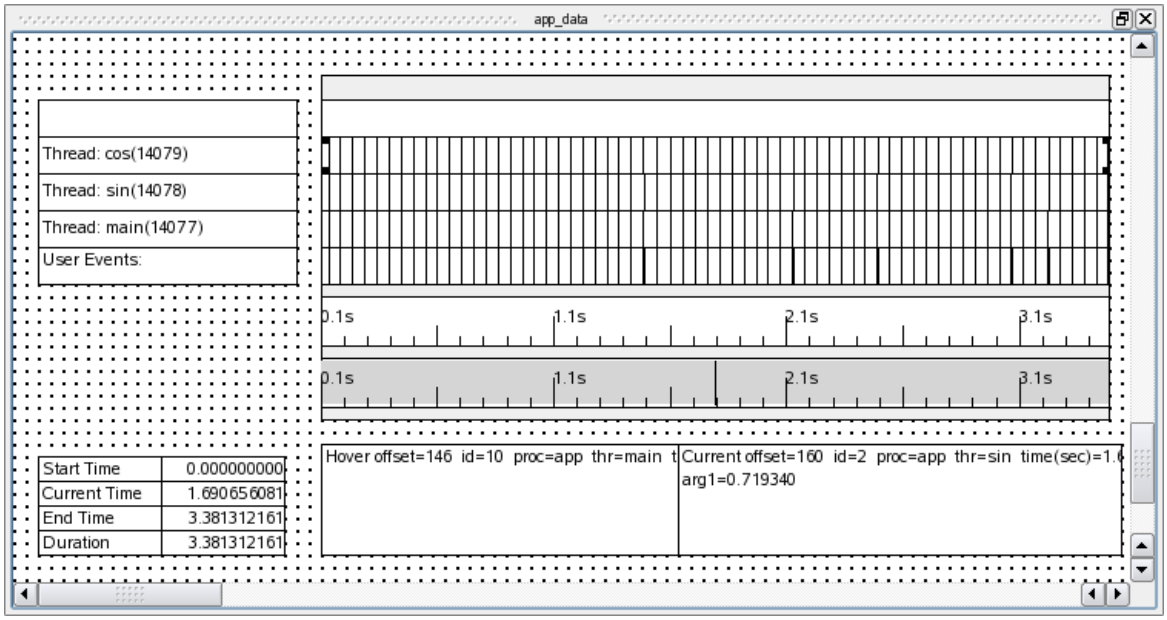


Figure 12-6. Timeline Editing

In edit mode, the background of the timeline turns into a grid. Objects can be created and inserted into the grid using the context menu.

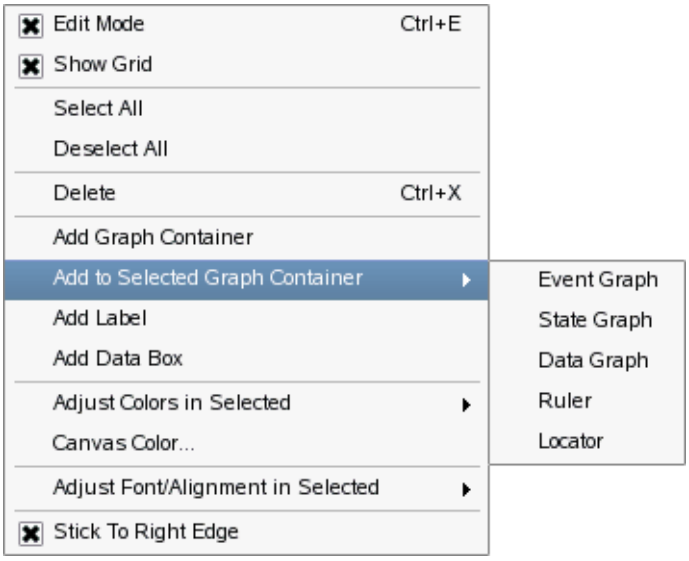


Figure 12-7. Timeline Context Menu

Most timeline objects must be inserted into a Graph Container. By default, a user timeline contains one large graph container consuming the center and largest portion of the timeline.

To insert an event graph, state graph, data graph, ruler, or locator into a graph container, select the graph container by clicking on it and then select the appropriate option from the context menu.

NOTE

If you cannot select the graph container because its edges are obscured by graphs within the container, click on any object in the container, then Shift+Click to select that container.

Once selected, the mouse cursor will change. Click inside the graph container and drag the mouse up or down and release the mouse button. The new object is inserted.

NOTE

Graph containers, and objects in general, can be resized using the mouse. Position the cursor over an edge or corner, wait for the cursor to change to a resizing cursor, then left click and drag to resize.

Double-click the new object to bring up its editing dialog, as described in the sections below.

Event Graph

An Event Graph displays vertical lines for each event that matches the criteria of the event graph.

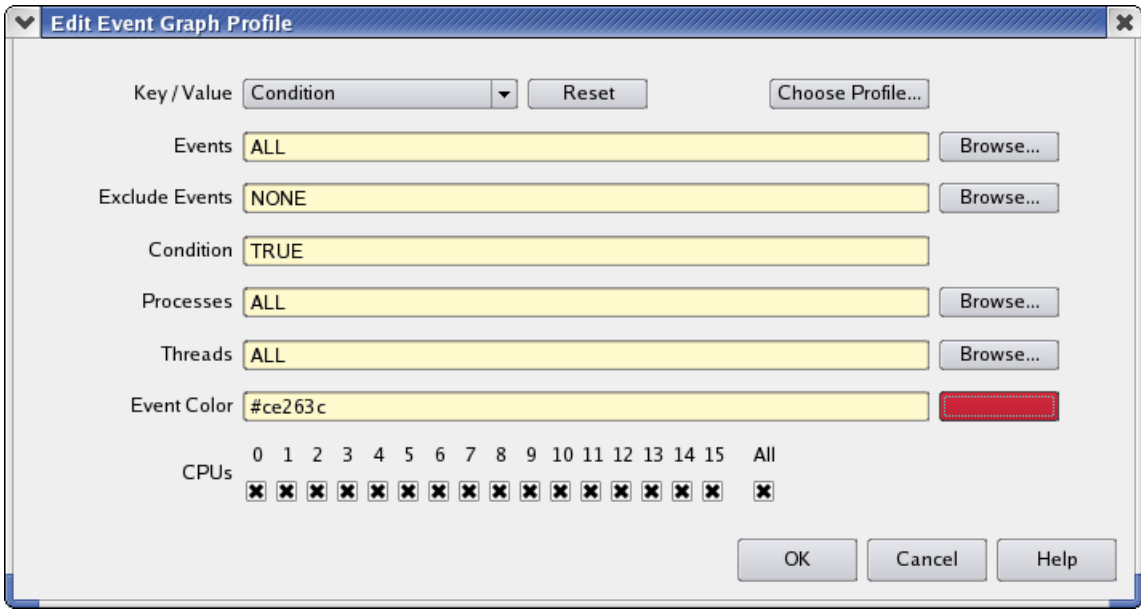


Figure 12-8. Edit Event Graph Profile Dialog

The definition of an event graph is essentially identical to defining a condition profile using the Profile Definition Panel.

Only events matching the conditions set within this dialog will be shown in the event graph.

Colors can be specified in the Event Color field by clicking on the color bar to the right of the text field and selecting a color from the Color Selection dialog or by entering in the text field a standard color name (see "Standard Color Names" on page 12-19) or RGB notation (i.e., **#***rrggbb* where *r*, *g* and *b* are hexadecimal characters representing the red, green and blue color components, respectively).

Additional adjustments can be made by selecting various options from the context menu when the event graph is selected.

State Graph

A State Graph is an Event Graph that can optionally display states as well.

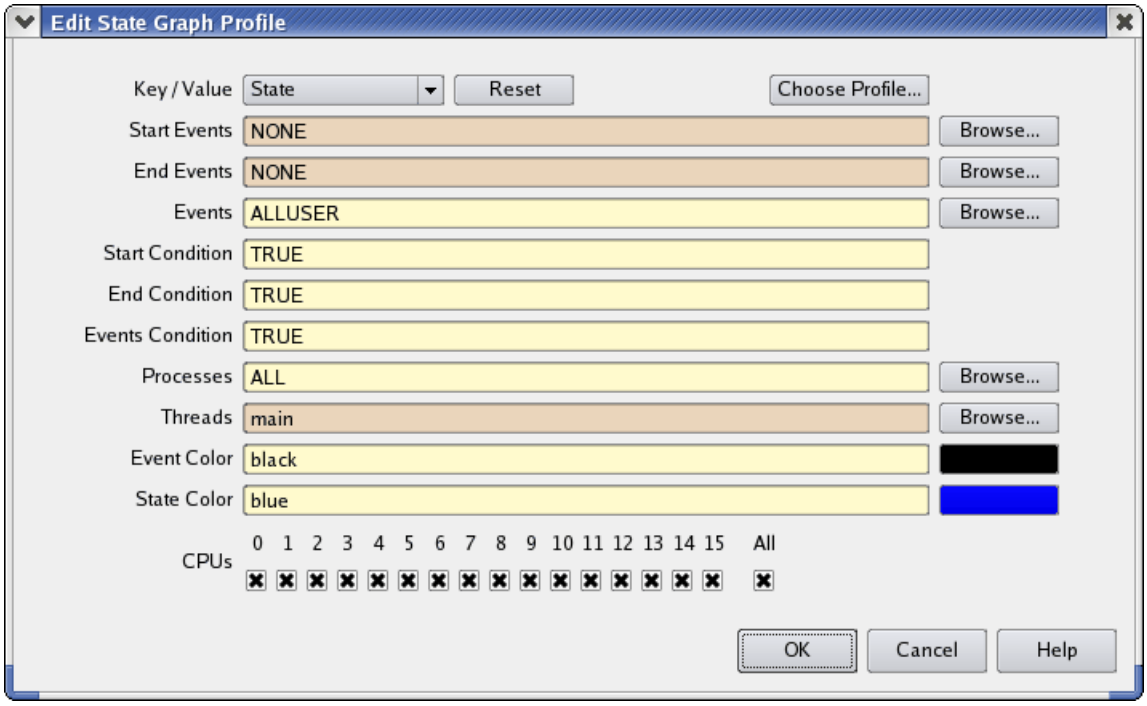


Figure 12-9. Edit State Graph Profile Dialog

The definition of a state graph is essentially identical to defining a state profile using the Profile Definition Panel, with the additional capability of selecting individual events to be displayed as in an Event Graph.

During the time in which a state is active, a solid bar appears in the lower vertical half of the state graph. Events as selected by the Events field in this dialog appear as vertical lines spanning the entire vertical space of the graph.

Colors can be specified in the Event Color and State Color fields by clicking on the color bar to the right of the text field and selecting a color from the Color Selection dialog or by entering in the text field a standard color name (see “Standard Color Names” on page 12-19) or RGB notation (i.e., **#***rrggbb* where *r*, *g* and *b* are hexadecimal characters representing the red, green and blue color components, respectively).

Additional adjustments can be made by selecting various options from the context menu when the state graph is selected.

Data Graph

A Data Graph is similar to a State Graph, except that a data block or line is shown in lieu of the solid state bar of a state graph. The height of the line or block indicates the value of the data.

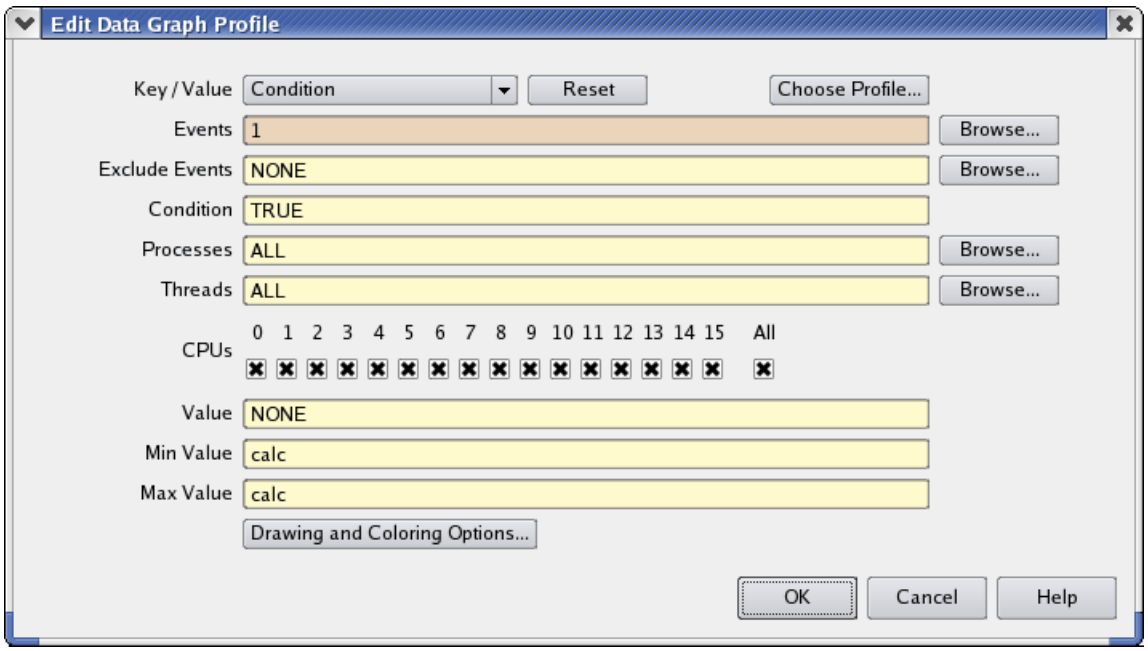


Figure 12-10. Edit Data Graph Profile Dialog

The definition of a data graph is essentially identical to defining a condition profile using the Profile Definition Panel, with the addition of three fields which define how the data is to be displayed.

Value

This field must be a valid NightTrace expression which defines a value. Typically this will be something simple like an argument associated with the events as defined in the Events field; e.g. `arg1`. See "Using Expressions" on page 16-1 for more information on expressions.

Min Value
Max Value

If set to CALC, NightTrace automatically calculates the minimum and/or maximum values of all data items matching the profile's criteria and adjusts the vertical scaling appropriately such that the largest data value consumes the entire vertical space of the graph and the smallest consumes a single pixel.

You may change the fields to specific values and NightTrace will adjust the scaling accordingly. Data values that fall outside the specified minimum or maximum values will be plotted as the minimum or maximum value specified, respectively.

Drawing and Coloring Options...

Pressing this button displays the Data Graph Options dialog that allows you to select the color of data values and their boundaries and select attributes which affect how the data graph is drawn.

Additional adjustments can be made by selecting various options from the context menu when the data graph is selected.

Data Graph Options Dialog

The Data Graph Options dialog is launched from the Edit Data Graph Profile dialog when the Drawing and Coloring Options... button is pressed.

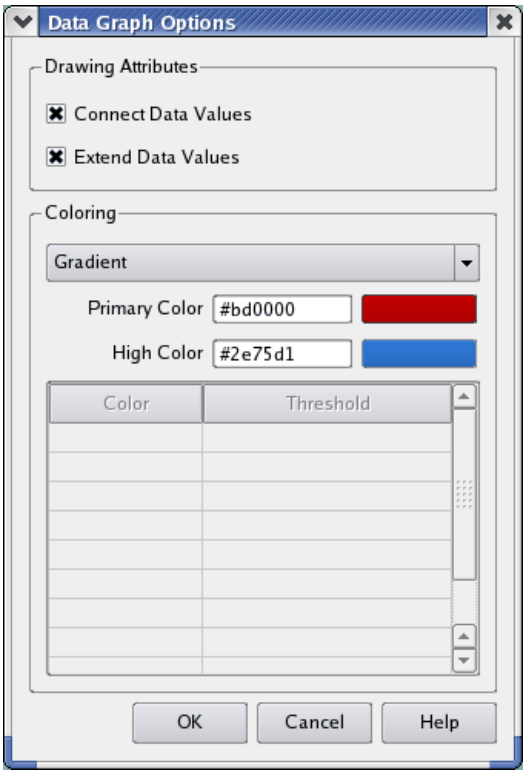


Figure 12-11. Data Graph Options Dialog

The dialog consists of two areas which control how data graphs are drawn and the colors used for the data values and boundaries.

Combining the various Drawing and Coloring options provides a wide variety of graph types, as shown in “Drawing and Coloring Examples” on page 12-16.

Drawing Attributes

Connect Data Values

This option draws a line between all consecutive data items. Each data item is drawn as a small point on the graph.

Extend Data Values

This option causes a polygon to be drawn, which extends from the X coordinate of the last data item up to the X coordinate of the current data item.

Coloring

The Coloring area defines the color mode used to draw the data graph and the colors associated with the mode selected from the dropdown:

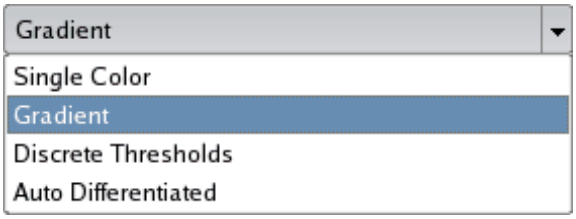


Figure 12-12. Data Graph Options Dialog Color Mode Selector

The color mode selector provides four options:

Single Color

In Single Color mode, a single color is used to draw all data values. The color is defined by the Primary Color item in the dialog.

Gradient

In Gradient mode, a linear color gradient is used to draw all data values and data value boundaries. The end-points of the gradient are defined by the Primary Color and High Color items in the dialog. The color gradient is strictly vertical, reflecting the value of each data item. Primary Color represents the smallest data value whereas High Color represents the largest data value.

Discrete Thresholds

In Discrete Thresholds mode, a set of colors is used to reflect various value thresholds of the data. An arbitrary number of thresholds can be entered, using the Color Thresholds table in the dialog.

The Primary Color is used as the default threshold -- the threshold matching all values not covered by specific thresholds entered in the table.

The portions of the data items and boundaries that are drawn that fall into each threshold will be of the corresponding threshold color.

Auto Differentiated

In Auto Differentiated mode, a unique color is randomly assigned to each data value encountered in the data graph. You cannot predict which color will be assigned to which data value, but once the color is shown it will remain associated with only that data value.

This option is not recommended for data sets which have a large range of values, since individual colors become hard to distinguish as the number of colors required increases dramatically.

An interesting application of this color mode combines its use with Extend Data Values and a strict application of graph Minimum and Maximum boundaries.

Consider a data set consisting of non-negative integers, such as the PID value of a set of processes. Setting the Minimum and Maximum graph boundaries in the Data Graph dialog to zero and one, respectively, combined with Extend Data Values and Auto Differentiated will cause a single block of data to be drawn for each data value of the same height, but with a unique color. Kernel display pages use this technique to show process activities on each CPU.

Primary Color

The Primary Color is used for the Single Color, Gradient, and Discrete Thresholds color modes.

A color may be selected by clicking on the color bar to the right of the text field and selecting a color from the Color Selection dialog or by entering in the text field a standard color name (see “Standard Color Names” on page 12-19) or RGB notation (i.e., **#***rrggbb* where *r*, *g* and *b* are hexadecimal characters representing the red, green and blue color components, respectively).

When a color is entered in the text field and the dialog focus moves away from the text field, the color bar is updated with the new color (unless it is invalid, in which case it turns black).

High Color

The High Color is only used with the Gradient color mode.

Colors may be selected by clicking on the color bar to the right of the text field and selecting a color from the Color Selection dialog or by entering in the text field a standard color name (see “Standard Color Names” on page 12-19) or RGB notation (i.e., **#***rrggbb* where *r*, *g* and *b* are hexadecimal characters representing the red, green and blue color components, respectively).

When a color is entered in the text field and the dialog focus moves away from the text field, the color bar to the text field is updated with the new color (unless it is invalid, in which case it turns black).

Color Thresholds

The Color Thresholds table is only used with the Discrete Thresholds color mode.

The table automatically expands as you enter individual thresholds.

Enter a color by clicking or entering a cell in the Color column. This launches a Color Selection dialog.

Enter a threshold value as an integer or floating-point numeric literal in the Threshold column by double-clicking in the cell or typing while positioned in the cell.

The value entered for a threshold is the inclusive lower bound of the threshold. The exclusive upper bound is defined by the closest threshold above it by value, not necessarily by visual position in the table. If no threshold exists, the upper bound extends to the maximum value that can be plotted.

Traverse the cells in the table by clicking with the mouse or using the arrow keys. Using the Tab key will cause the focus to leave the table.

Remove cells by selecting the cells to be removed and pressing the Delete key (or Crtl+X).

Thresholds are automatically sorted in ascending order by NightTrace before and after the dialog is shown.

The Primary Color is used for the default threshold, which matches all values lower than the lowest threshold entered in the table.

Drawing and Coloring Examples

Figure 12-13 shows several different data graphs reflecting the same data, but using different combinations of Drawing and Coloring attributes.

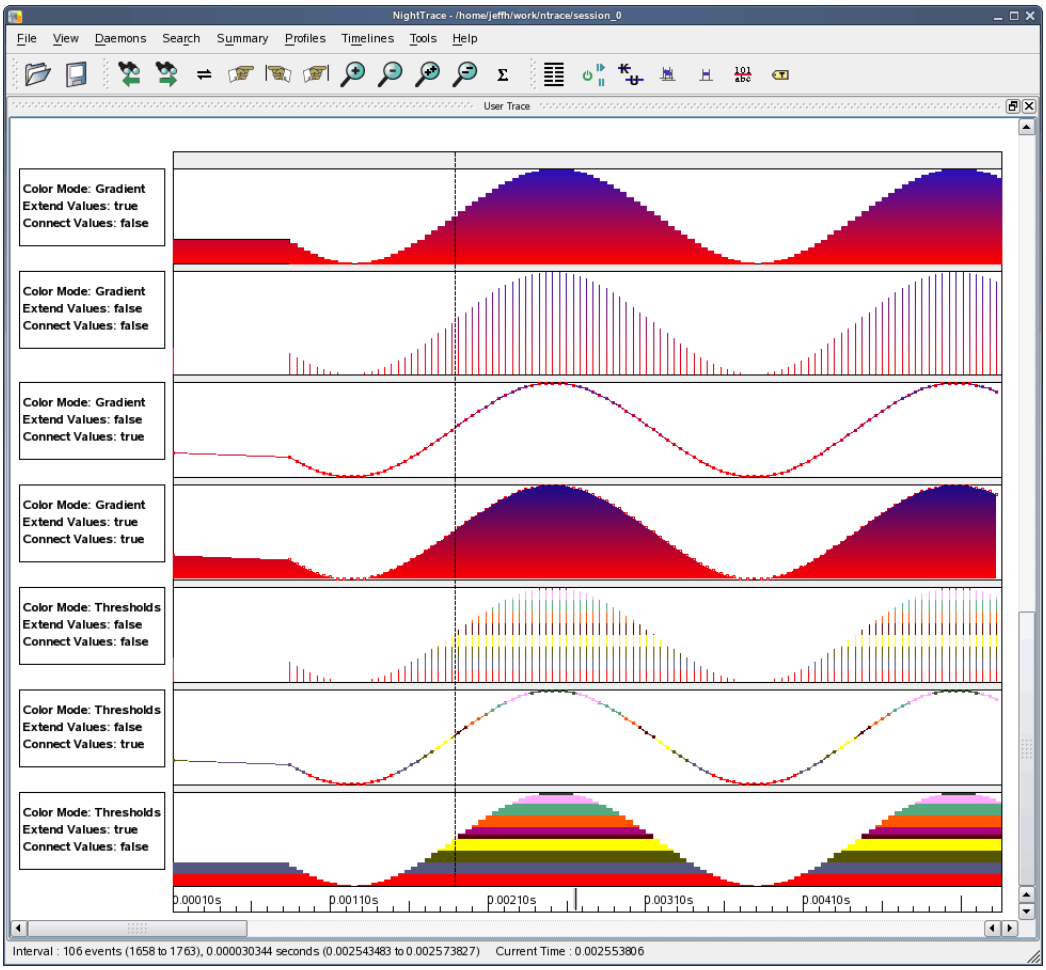


Figure 12-13. Data Graph Examples

Color Selection Dialog

The Color Selection Dialog aids you in selecting a color by allowing you to select from a list of basic or customized colors, enter RGB values, or select a color from a spectrum.

It is launched when clicking on a colored button to the right of a color selection text field, or when clicking in cells in the Color column of the Color Thresholds table.

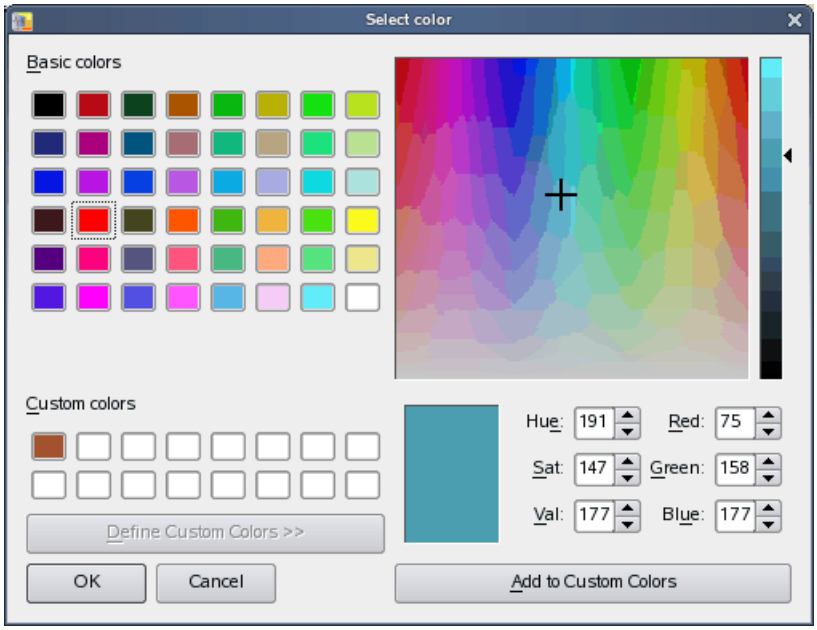


Figure 12-14. Color Selection Dialog

When using the mouse to select a color from the spectrum, be sure to choose an Alpha value from the slider at the right-hand side of the dialog.

A common error is to click in the spectrum area and click on OK, expecting to get the exact color associated with your mouse click in the spectrum, but effectively getting black instead due to the Alpha setting. The color in the spectrum is modified by the Alpha value associated with the vertical slider setting. The actual color you are selecting is always shown in the medium-sized rectangle beneath the lower-left corner of the spectrum.

Standard Color Names

NightTrace supports the standard color names shown in Table 12-1.

Table 12-1. Standard Color Names

aliceblue		darkslategray		lightpink		paleturquoise	
antiquewhite		darkslategrey		lightsalmon		palevioletred	
aqua		darkturquoise		lightseagreen		papayawhip	
aquamarine		darkviolet		lightskyblue		peachpuff	
azure		deeppink		lightslategray		peru	
beige		deepskyblue		lightslategrey		pink	
bisque		dimgray		lightsteelblue		plum	
black		dimgrey		lightyellow		powderblue	
blanchedalmond		dodgerblue		lime		purple	
blue		firebrick		limegreen		red	
blueviolet		floralwhite		linen		rosybrown	
brown		forestgreen		magenta		royalblue	
burlywood		fuchsia		maroon		saddlebrown	
cadetblue		gainsboro		mediumaquamarine		salmon	
chartreuse		ghostwhite		mediumblue		sandybrown	
chocolate		gold		mediumorchid		seagreen	
coral		goldenrod		mediumpurple		seashell	
cornflowerblue		gray		mediumseagreen		sienna	
cornsilk		grey		mediumslateblue		silver	
crimson		green		mediumspringgreen		skyblue	
cyan		greenyellow		mediumturquoise		slateblue	
darkblue		honeydew		mediumvioletred		slategray	
darkcyan		hotpink		midnightblue		slategrey	
darkgoldenrod		indianred		mintcream		snow	
darkgray		indigo		mistyrose		springgreen	
darkgreen		ivory		moccasin		steelblue	
darkgrey		khaki		navajowhite		tan	
darkkhaki		lavender		navy		teal	
darkmagenta		lavenderblush		oldlace		thistle	
darkolivegreen		lawngreen		olive		tomato	
darkorange		lemonchiffon		olivedrab		turquoise	
darkorchid		lightblue		orange		violet	
darkred		lightcoral		orangered		wheat	
darksalmon		lightcyan		orchid		white	
darkseagreen		lightgoldenrodyellow		palegoldenrod		whitesmoke	
darkslateblue		lightgray		palegreen		yellow	

Interval Ruler

You can add an Interval Ruler to a graph container using the Ruler option of the Add to Selected Graph Container sub-menu of the timeline's context menu.

Global Ruler

You can add a Global Ruler to a graph container using the Locator option of the Add to Selected Graph Container sub-menu of the timeline's context menu.

Label

Labels are static text areas that can be placed anywhere within a timeline. They do not have to be inserted into a graph container.

You can add a label by using the Add Label option of the timeline's context menu.

Once added, double-click the label to set its text.

Once defined, you can adjust attributes of the label by selecting various options from the context menu when the label is selected, for example:

Adjust Font/Alignment in Selected

This menu item allows you to select a font for the label, and to adjust its vertical and horizontal alignment.

Adjust Colors in Selected

This menu item allows you to select the color of the text and the color of the label's background.

Data Box

A Data Box is a dynamic label that can be placed anywhere in a timeline. The value displayed in the box is dependent on the current timeline.

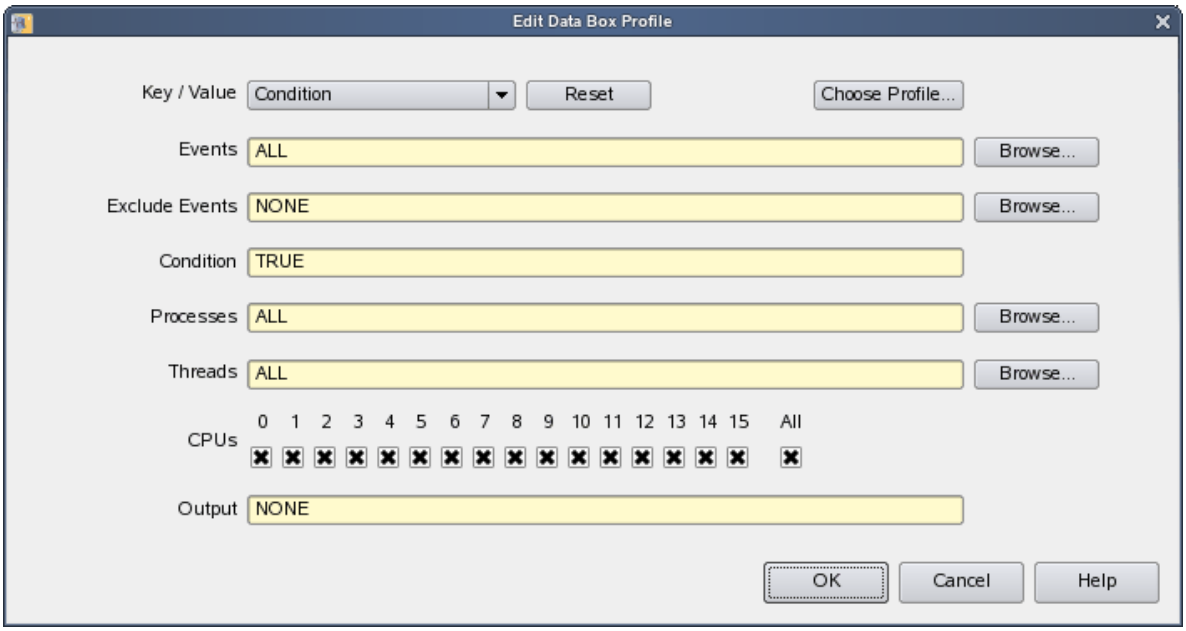


Figure 12-15. Edit Data Box Profile

The definition of a Data Box is essentially identical to defining a condition profile using the Profile Definition Panel, with the addition of the following field:

Output

This field must be a valid NightTrace string expression. Typically, it involves use of the `format()` function. For example:

```
format("The current value is: %f", arg_dbl())
```

See "Using Expressions" on page 16-1 for more information.

Once defined, you can adjust the box by selecting various options from the context menu when the data box is selected; for example:

Adjust Font/Alignment in Selected

This menu item allows you to select font for the text to be displayed and to adjust its vertical and horizontal alignment.

Adjust Colors in Selected

This menu item allows you to select the color of the text and the color of the label's background.

13
Profiles Panels

Profiles include any condition or state you use within a NightTrace session, including those used in search and summary operations.

In NightTrace, a condition is the "logical and" of several criteria such as event codes, processes, and threads. Conditions may be used to examine matching events of interest.

A state profile is a combination of two conditions which identify the start and end requirements of a state. All other profiles are simply condition profiles, although they can be as complex as you need them to be.

Profiles can be used in:

- searches
- summaries
- graphs

Profiles are managed using the Profile Status List and the Profile Definition panels.

Profile Definition Panel

This panel allows you to define new profiles using drop-down option lists for commonly requested conditions and states. Profiles can be further customized providing you complete control over detailed profile conditions.

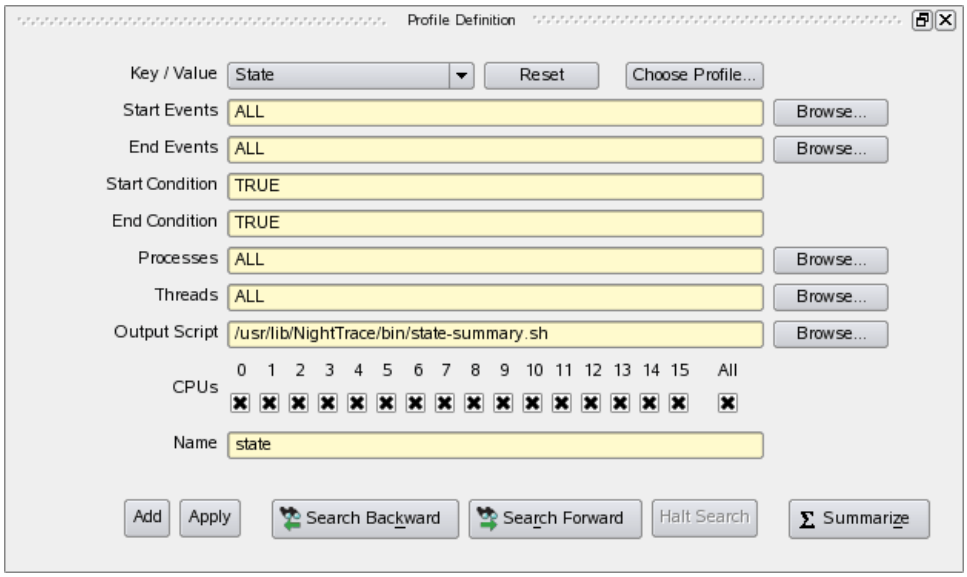


Figure 13-1. Profile Definition Panel

Key/Value

The Key/Value option list provides a starting point for profile definition. Selecting items from the option list populates the individual condition fields below with the values and expressions required to specify the key (and value) you have selected.

The option list provides the following items:

Condition

This option populates the condition fields to create a condition profile which will match any event, unconditionally. It is useful when you wish to manually enter conditions starting from a clean template.

State

This option populates the condition fields to create a state profile which starts on any event and ends on any event. It is useful when you wish to manually enter state conditions starting from a clean template.

System Call All Events
System Call Enter Events
System Call Exit Events
System Call State

These options desensitized if kernel trace data is not loaded.

These options populate the condition fields such that the profile detects the existence of a specific system call, as indicated by the specific option selected. After selecting one of these options, a system call list will launch allowing you to select an individual system call.

Selecting System Call All Events will match events representing the entry, suspension, resumption, and exit of a system call.

Selecting System Call Enter Events or System Call Exit Events will match events representing entry and resumption of a system call, or suspension and exit, respectively.

Selecting System Call State defines a state which begins when a system calls is entered or resumed, and terminates when the system call is suspended or exits.

When a specific system call is selected, the name of the system call will appear in a read-only text field beneath the Key/Value option list. The specific system call associated with the profile can be changed by pressing the Values… button and selecting a different value from the list.

NOTE

Multiple system calls may be selected from the Key/Value pop-up menu.

Exception All Events
Exception Enter Events
Exception Exit Events
Exception State

These options desensitized if kernel trace data is not loaded.

These options populate the condition fields such that the profile detects the existence of a specific machine exception, as indicated by the specific option selected. After selecting one of these options, an exception list will launch allowing you to select an individual exception.

Selecting Exception All Events will match events representing the entry, suspension, resumption, and exit of an exception.

Selecting Exception Enter Events or Exception Exit Events will match events representing entry and resumption of an exception, or suspension and exit, respectively.

Selecting Exception State defines a state which begins when an exception is entered or resumed, and terminates when the exception is suspended or exits.

When a specific exception is selected, the name of the exception will appear in a read-only text field beneath the Key/Value option list. The specific exception associated with the profile can be changed by pressing the Values… button and selecting a different value from the list.

NOTE

Multiple exceptions may be selected from the Key/Value pop-up menu.

Interrupt All Events
Interrupt Enter Events
Interrupt Exit Events
Interrupt State

These options desensitized if kernel trace data is not loaded.

These options populate the condition fields such that the profile detects the existence of a specific machine interrupt, as indicated by the specific option selected. After selecting one of these options, an interrupt list will launch allowing you to select an individual interrupt.

Selecting Interrupt All Events will match events representing the entry, suspension, resumption, and exit of an interrupt.

Selecting Interrupt Enter Events or Interrupt Exit Events will match events representing entry and resumption of an interrupt, or suspension and exit, respectively.

Selecting Interrupt State defines a state which begins when an interrupt is entered and terminates when the interrupt exits.

When a specific interrupt is selected, the name of the interrupt will appear in a read-only text field beneath the Key/Value option list. The specific interrupt associated with the profile can be changed by pressing the Values… button and selecting a different value from the list.

NOTE

Multiple interrupts may be selected from the Key/Value pop-up menu.

Tagged Events

This option populates the condition fields such that the profile detects the event associated with the tag that you select from the list that is launched when choosing this option.

When a specific tag is selected, the name of the tag will appear in a read-only text field beneath the Key/Value option list. The specific tag associated with the profile can be changed by pressing the Values… button and selecting a different value from the list.

If no tagged events exist, this menu option is desensitized.

NOTE

Multiple tags may be selected from the Key/Value pop-up menu.

Choose Profile...

You can select from previously-defined profiles using the Choose Profile… button.

Selecting an entry from the list displayed by this button populates the Profile Definition panel with the conditions associated with that profile. The current profile becomes the profile you selected. Subsequent changes will be applied to the profile if you press the Apply, Search/Close, or Summarize buttons. A new profile will be created if you press the Add button.

Alternatively, when checking the Import by Reference checkbox in the Choose Profile dialog, the Profile Definition panel will be populated with a condition that references the selected profile. This technique allows you to add additional conditions to the selected profile while preserving the named association. Thus subsequent changes to the selected profile will be reflected in the new profile you create.

After choosing a Key/Value pair or previously defined profile using the Choose Profile… button, you can further customize the condition or state by using the individual text fields and selection lists in the dialog.

Any customized changes which are subsequently made appear in the criteria text fields with a salmon-colored background. Pressing the Reset button restores the default conditions that were populated when you selected the profile.

Events
Start Events
End Events

The Events, Start Events and End Events criteria allows you restrict the condition to events listed in the text fields. Values in the text fields are required to be a comma-separated list of numeric event numbers or ranges or event names. The Browse… buttons to the right of the text fields allows you to select from a list of known event names. The values ALL, ALLADA, ALLKERNEL, and ALLUSER are special entries referring to classes of events, as indicated by their name.

Start Events and End Events are only shown for state profiles whereas Events is only shown for condition profiles. Start Events and End Events refers to events which are candidates for the beginning or end of a state, respectively. Events refers to all events.

Exclude Events

Exclude Events allows you restrict the condition to events that are not listed in the text field. It is only shown for condition profiles.

Values in the text field are required to be a comma-separated list of numeric event numbers or ranges or event names. The Browse… button to the right of the text field allows you to select from a list of known event names. The value NONE is a special entry referring to null set of events, which means that no events are excluded.

Condition
Start Condition
End Condition

The Condition, Start Condition, and End Condition criteria allows you restrict the profile using NightTrace's expression language. Values in the text fields are required to be a boolean NightTrace expressions whose syntax is roughly that of the C language, with built-in functions for accessing attributes of events. See "Using Expressions" on page 16-1 for more information on expression syntax and semantics.

Start Condition and End Condition are only shown for state profiles whereas Condition is only shown for condition profiles. Start Condition and End Condition refers to the conditions which must be met for the beginning or end of a state, respectively, whereas Condition applies globally to the profile.

Processes

The Processes criterion allows you restrict the condition to events generated by processes that are specified in the text field.

Values in the text field are required to be a comma-separated list of process names or PIDs (see **`getpid(2)`** and **`gettid(2)`**). The Browse… button to the right of the text field allows you to select from a list of known processes.

NOTE

All Linux threads within the same program share the same PID value. For trace events generated with the NightTrace Logging API, the value logged as the process identifier is the common PID. For kernel events, the value logged for the process identifier is the thread's TID (see **`gettid(2)`**).

If multiple processes have the same name (perhaps two unrelated programs both called **`a.out`**) selecting that name from the list or placing that text in the text field

will match both processes. Similarly, for multi-threaded processes, the specified process name will match all threads within the process.

Placing a process name in the Processes list is equivalent to adding a condition restriction using the following NightTrace expression:

```
process_name == "a.out"
```

Threads

The Threads criterion allows you restrict the condition to events generated by threads that are specified in the text field.

Values in the text field are required to be a comma-separated list of thread IDs (see **gettid(2)**). The Browse… button to the right of the text field allows you to select from a list of known threads by name. This list is only available when user trace data from registered threads is loaded. See "Threads and Logging" on page 2-33 for more information.

If multiple threads with the same name exist, specifying the thread name will match all such threads.

Placing a thread name in the Threads list is equivalent to adding to adding a condition restriction using the following NightTrace expression:

```
thread_name == "mythread"
```

Nodes

The Nodes criterion allows you restrict the condition to events generated on the systems that are specified in the text field.

Values in the text field are required to be a comma-separated list of system names (see **hostname(1)**). The Browse… button to the right of the text field allows you to select from a list of known hosts present in the loaded trace data sets by name.

Use of the Nodes condition is only useful when capturing and analyzing data from multiple systems using the Real-time Clock and Interrupt Module (RCIM) as a synchronized timing source. See the *Real-Time Clock and Interrupt Module User's Guide* (0891082) for more information.

Placing a node name in the Nodes list is equivalent to adding to adding a condition restriction using the following NightTrace expression:

```
node_name == "a.out"
```

Output Script

This text field does not impose a constraint on the profile. It allows you to specify an alternative shell script that is executed for summary operations. By default, the following scripts are executed for condition and state profile summaries, respectively:

- **/usr/lib/NightTrace/bin/event-summary.sh**

- **/usr/lib/NightTrace/bin/state-summary.sh**

All script output generated to *stdout* will be displayed in the Profiles Result panel which is automatically created when a summary is executed for a new profile. Output from *stderr* is not captured.

Summary data is passed to the specified script via environment variables. See "Summary Script Environment Variables" on page 13-11 for more information.

The path to the summary output script is saved as part of a NightTrace session and can be utilized in subsequent **ntrace** invocations, including batch mode summary execution via command line options.

CPUs

The CPUs selector area allows you to place CPU restrictions on the profile. Use the checkboxes to select the CPUs of interest.

Name

The Name text field defines the name of the profile. The profile's name is automatically set when selecting a previously-defined profile or when creating a new profile. You can change the name by typing in a modified name in the text field. Changing the name of a profile does not, in and of itself, create a new profile. A new profile is created if you press the Add button. Pressing the Apply, Search/Close, or Summaries buttons applies the name change (and all other outstanding profiles changes) to the current profile as well as executes the associated action, if any.

Control Buttons

The buttons at the bottom of the panel operate on the profile as defined by the remainder of the panel.

Add

The Add button creates a new profile based on the conditions in the Profile Definition panel. If another profile with the same name already exists, the name of the new profile is automatically adjusted to be unique by appending a numeric value to the name.

Apply

The Apply button modifies an existing profile based on the conditions in the Profile Definition panel. If the profile did not previously exist, it adds the profile.

Search Backward

Executes a backward search for the selected profile.

Search Forward

Executes a forward search for the selected profile.

Halt Search

Halts a currently active search.

Summarize

The Summarize button executes a summary action based on the current profile.

Summaries can also be executed by pressing the Summary icon on the tool bar or selecting the Summarize option from the Summary menu.

See “Summarizing Statistical Information” on page 13-10 for more information.

Summarizing Statistical Information

A variety of statistics are available for summaries of condition and state profiles.

Condition Summaries

The following statistics are provided for condition profile summaries:

- The number of matches summarized
- The minimum time gap between matches and the ordinal trace event number (offset) where it began
- The maximum time gap between matches and the ordinal trace event number (offset) where it began
- The average time gap between matches

State Summaries

The following statistics are generated for state profile summaries:

- The number of matches summarized
- The minimum time gap between matches and the ordinal trace event number (offset) where it began
- The maximum time gap between matches and the ordinal trace event number (offset) where it began
- The average time gap between matches
- The sum of the time gaps between matches
- The minimum time duration of a match and the ordinal trace event number (offset) where it began
- The maximum time duration of a match and the ordinal trace event number (offset) where it began
- The average time duration of a match
- The sum of the time durations of matches

Summary Scripts

Summary results are printed by invoking summary scripts to display the statistical information. By default, NightTrace provides an event summary and a state summary script that print the statistics as described above.

User-define scripts may be used in place of the default scripts. See “Output Script” on page 13-7 for more information on specifying user-defined scripts.

Summary Script Environment Variables

The following summary environment variables are passed to summary scripts

Table 13-1. Summary Script Environment Variables

Variable	Meaning
NT_SUM_TYPE	Contains text describing the type of summary: “Event Summary” or “State Summary”.
NT_SUM_NUM	The number of occurrences of the state or event, expressed in decimal integer format.
NT_SUM_MIN_GAP	The minimum gap between occurrences of the state or event, expressed in seconds in decimal floating point format.
NT_SUM_MAX_GAP	The maximum gap between occurrences of the state or event, expressed in seconds in decimal floating point format.
NT_SUM_AVG_GAP	The average gap between occurrences of the state or event, expressed in seconds in decimal floating point format.
NT_SUM_TOTAL_GAP	The total time for all gaps between occurrences of the state or event, expressed in seconds in decimal floating point format.
NT_SUM_MIN_GAP_OFFSET	The offset at which the minimum gap between occurrences of the state or event occurred expressed in decimal integer format.
NT_SUM_MAX_GAP_OFFSET	The offset at which the maximum gap between occurrences of the state or event occurred expressed in decimal integer format.
NT_SUM_MIN_DURATION	For states, the minimum state duration expressed in seconds in decimal floating point format.
NT_SUM_MAX_DURATION	For states, the maximum state duration expressed in seconds in decimal floating point format.
NT_SUM_AVG_DURATION	For states, the average state duration expressed in seconds in decimal floating point format.

Table 13-1. Summary Script Environment Variables

Variable	Meaning
NT_SUM_TOTAL_DURATION	For states, the total of all state durations, expressed in seconds in decimal floating point format.
NT_SUM_MIN_DURATION_OFFSET	For states, the offset at which the minimum state duration occurred, expressed in decimal integer format.
NT_SUM_MAX_DURATION_OFFSET	For states, the offset at which the maximum state duration occurred, expressed in decimal integer format.

Profile Status List Panel

The Profile Status List panel displays all profiles in the current NightTrace session.

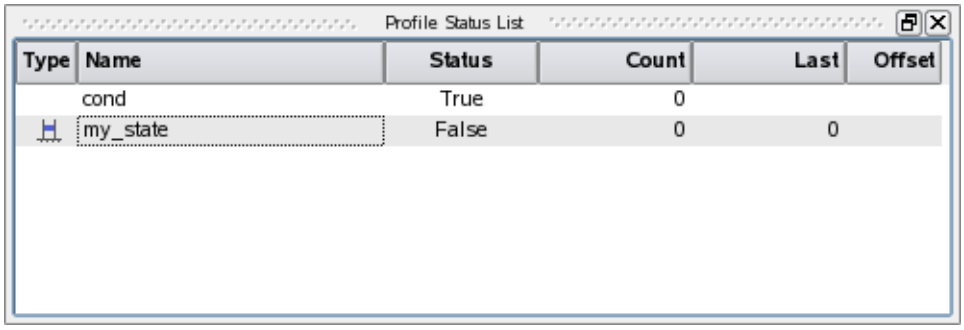


Figure 13-2. Profile Status List Panel

Profile Status List Table

The profiles are displayed in a table with the following columns:

Type

This column displays a state icon for state profiles; otherwise nothing is displayed.

Name

This column displays the profile's name.

Status

This column indicates whether the event at or immediate previous to the current timeline satisfies the conditions of the profile.

Count

This column displays a count of the number of instances of events that satisfy the conditions of the profile.

Last

This column displays the last event offset before the current timeline which satisfied the conditions of the profile.

Offset

This column displays the last event offset that concluded the profile's state -- this is only valid for states.

Context Menu

The Profile Status List panel's context menu is shown below.

Figure 13-3. Profile Status List Panel Context Menu

New Profile

This option raises the page which contains a Profile Definition panel, or creates such a panel on the current page if no such panel already exists.

Delete

This option deletes the profile definitions currently selected in the panel.

Search Forward

This option executes a forward search for the currently selected profile.

Search Backward

This option executes a backward search for the currently selected profile.

Search Forward within Timeline Interval

This option executes a forward search for the currently selected profile; the range of events to search is constrained by the current Timeline interval.

Search Backward within Timeline Interval

This option executes a backward search for the currently selected profile; the range of events to search is constrained by the current Timeline interval.

Summarize

This option executes a summary action on the currently selected profile.

Summarize within Timeline Interval

This option executes a summary action on the currently selected profile; the range of events to summary is constrained by the current Timeline interval.

Move Up

This option moves the currently selected profiles one position towards the beginning of the table.

Move Down

This option moves the currently selected profiles one position towards the end of the table.

Display Fields

This option displays a sub-menu which allows you to select which columns are visible within the table.


Type
Name
Status
Count
Last
Offset

14
Event Descriptions Panel

The Event Descriptions panel presents a table with a row for each known event ID. The table describes the event name and description associated with each event ID.

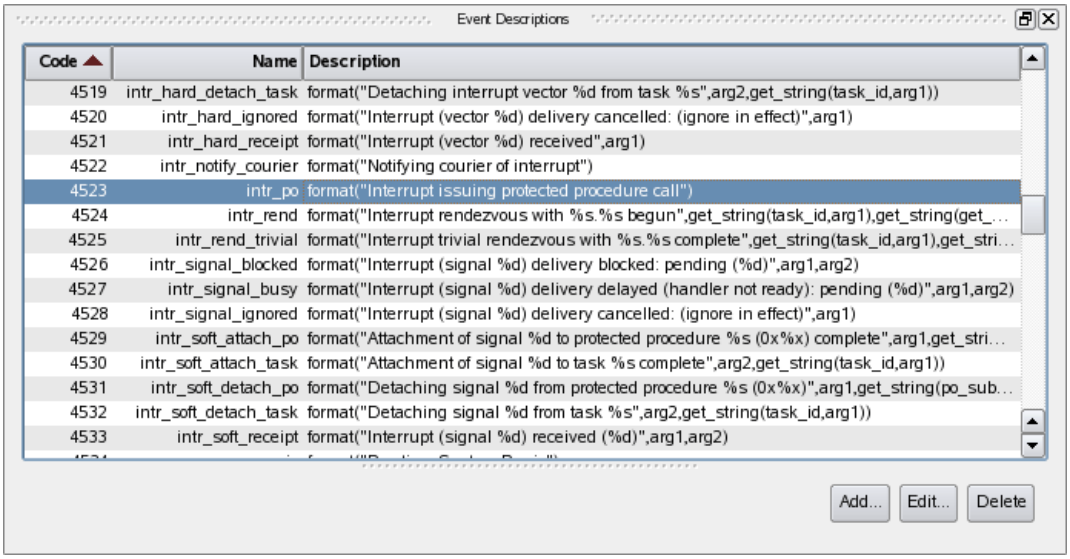


Figure 14-1. Event Descriptions Panel

The table can be sorted by clicking on a column header. Subsequent clicks on a column header cell that is already defined as the sort key (as indicated by the dark-red chevron), causes the sort direction to reverse.

The table consists of the following columns.

Code

This column contains the event ID of interest.

Name

This column defines the textual name that will be displayed in lieu of the event ID.

Description

This column describes the format of the textual description used for the event.

Pressing the Add... or Edit buttons launches the Event Description dialog which allows you to change these values.

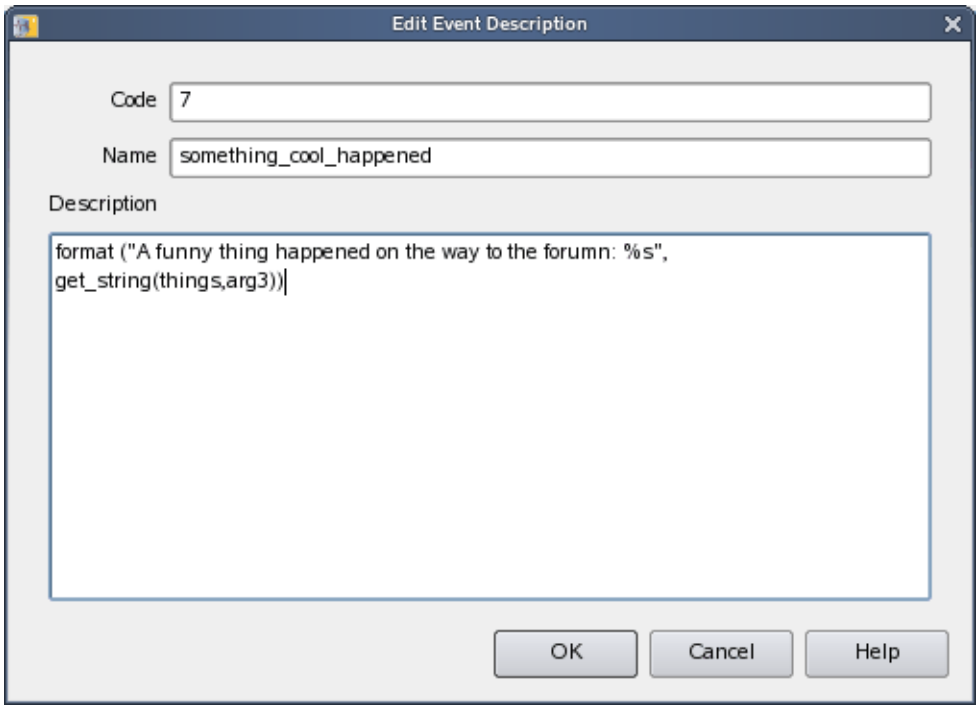


Figure 14-2. Event Description Dialog

The Description field allows you to use the NightTrace `format()` function to define a (possibly complex) textual description of the event and its arguments.

15
Tags List Panel

The Tags List panel presents a table of all tagged events in the current NightTrace session.

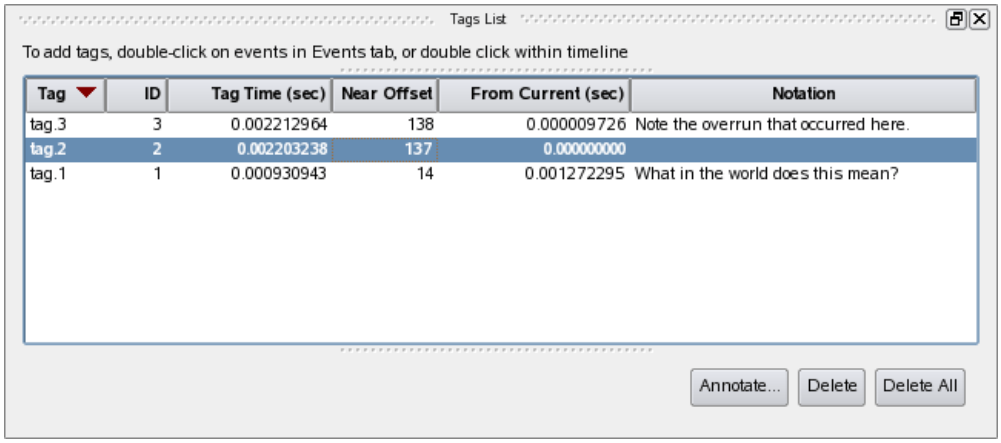


Figure 15-1. Tags List Panel

Tags are a convenient mechanism of identifying an event or time of interest.

Tags appears as small yellow notes with the tag's number on the ruler of Timelines.

Tags are saved as part of NightTrace sessions, so they can be useful in quickly locating an event of interest in subsequent execution of NightTrace on the same data set.

The notation capability allows you to add explanatory text for a tag and to share it with others by saving the session and directing another user to look for a specific tag name.

You can search for tags by name using the Profile Search dialog (see "Profiles Panels" on page 13-1 for more information on profile searching).

Creating Tags

You can create a tag using one of the following three methods:

1. Double-click on any row in an Events panel; the tag will be associated with the time of the event whose row you double-clicked.
2. Double-click on any event in an EventGraph in a Timeline; the tag will be associated with the time of the event you double-clicked.

3. Double-click on a ruler in a Timeline -- the tag will be associated with the time associated with the location you clicked in the ruler.

Tags List Table

Clicking on a row in the Tags List table causes the current timeline to be moved to the time associated with the tag.

The Tags List table consists of the following columns:

Tag

This column shows the name of the tag.

ID

This column shows the tag's integer ID value.

Tag Time

This column shows the time of the tag.

Near Offset

This column shows the ordinal offset of the nearest event.

From Current

This column shows the time between the tag and the current timeline.

Since the current timeline is always moved to the time associated with the tag you click in the table, its From Current value will often be zero (unless you change the location of the current timeline with some other operation -- e.g. executing a search or clicking in a Timeline panel).

Notation

The notation field is free-form text which you can provide.

Context Menu

The Tags List panel context menu is shown below.

Figure 15-2. Tags List Panel Context Menu

Annotate...

This option opens a simple dialog which lets you add or change the notation associated with the selected tag. This option is disabled if multiple tags are currently selected.

Delete

This option deletes all currently selected tags.

Delete All

This option deletes all tags in the current session.

Display Fields

This option displays a sub-menu which allows you to select which columns are visible within the table.

Control Buttons

The Annotate… and Delete buttons operate on the currently selected tags in the table (the Annotate… button is disabled if more than one tag is selected).

The Delete All button deletes all tags from the current session.

16
Using Expressions

Overview

NightTrace allows you to use expressions to aid in the analysis of trace data.

NightTrace expressions are comprised of a combination of *operators* and *operands* and can evaluate to numbers, strings, or boolean values.

See “Operators” on page 16-1 for a list of valid operators and “Operands” on page 16-1 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 cast operations for the following supported data types (where the parentheses are required):
  - (long long)
  - (long double)
  - (unsigned long)
  - (unsigned long long)

NightTrace operators follow the operator precedence rules of the C programming language.

Operands

Operands include:

- “Constants” on page 16-2

- "Functions" on page 16-4
- "Profile References" on page 16-193 (in functions only)

Operand types are largely based on the C programming language and include:

- integer
- long integer
- long long integer
- double-precision floating point
- long double-precision floating point
- character
- string
- boolean
- bit fields

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.

Standard C decimal floating point literals are supported and have the following syntax:

fore . *aft*[`E` |`e`[`+` |`-`]*exp*]

fore . *aft*

any combination of decimal digits 0 through 9

`E` or `e`

can optionally precede an optional sign and exponent

+ or -

optional sign

exp

a decimal number specifying the power of 10 to which *fore* . *aft* is multiplied

Alternatively, floating point literals following the C99 standard are also supported and have the following syntax:

`0x`*fore* . *aft* [`P` | `p`[`+` | `-`]*exp*]

`0x`

defines this as a hexadecimal literal

fore . *aft*

any combination of hexadecimal digits 0 through 9, a through f, or A through F.

`P` or `p`

can optionally precede an optional sign and exponent

+ or -

optional sign

exp

a decimal number specifying the power of 2 to which *fore* . *aft* is multiplied

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 16-1 shows units and suffixes for time constants.

Table 16-1. 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:

- "String Functions" on page 16-16
- "Trace Event Functions" on page 16-18
- "State Functions" on page 16-60
- "Offset Functions" on page 16-138
- "Summary Functions" on page 16-177
- "Format and Table Functions" on page 16-184

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 *offset* (or ordinal trace event number). See "Offset Functions" on page 16-138.

`start_`

Functions with this prefix provide information about the *start event* of the *most recent instance of a state*. See "Start Functions" on page 16-60.

`end_`

Functions with this prefix provide information about the *end event* of the *last completed instance of a state* See "End Functions" on page 16-97.

`state_`

Functions with this prefix provide information about instances of states. See "Multi-State Functions" on page 16-134.

`event_`

Functions with this prefix provide information about instances of events. See "Multi-Event Functions" on page 16-58.

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 16-1 contains a complete list of functions sorted by general catagories. For an alphabetic list of all functions, refer to the Index.

Table 16-1. NightTrace Functions

Syntax	Return Type
`strcmp (`*s1*`, `*s2*`)` `strncmp (`*s1*`, `*s2*`, `*n*`)`	An integer indicating less than, equal to, or greater than zero as *s1*, or the first *n* bytes thereof, is compared to *s2*.
`id` [(`[`*PR*`]`)] `start_id` [(`[`*PR*`]`)] `end_id` [(`[`*PR*`]`)] `offset_id` (*offset_expr*`)`	The integer *trace event ID*.
`arg[`*N*`]` [(`[`*PR*`]`)] `start_arg[`*N*`]` [(`[`*PR*`]`)] `end_arg[`*N*`]` [(`[`*PR*`]`)] `offset_arg[`*N*`]` (*offset_expr*)	The integer *trace event argument*.
`arg[`*N*`]_dbl` [(`[`*PR*`]`)] `start_arg[`*N*`]_dbl` [(`[`*PR*`]`)] `end_arg[`*N*`]_dbl` [(`[`*PR*`]`)] `offset_arg[`*N*`]_dbl` (*offset_expr*)	The double-precision floating point *trace event argument*.
`arg[`*N*`]_long` [(`[`*PR*`]`)] `start_arg[`*N*`]_long` [(`[`*PR*`]`)] `end_arg[`*N*`]_long` [(`[`*PR*`]`)] `offset_arg[`*N*`]_long` (*offset_expr*)	The long integer *trace event argument*.
`arg[`*N*`]_long_dbl` [(`[`*PR*`]`)] `start_arg[`*N*`]_long_dbl` [(`[`*PR*`]`)] `end_arg[`*N*`]_long_dbl` [(`[`*PR*`]`)] `offset_arg[`*N*`]_long_dbl` (*offset_expr*)	The long double-precision *trace event argument*.
`arg[`*N*`]_long_long` [(`[`*PR*`]`)] `start_arg[`*N*`]_long_long` [(`[`*PR*`]`)] `end_arg[`*N*`]_long_long` [(`[`*PR*`]`)] `offset_arg[`*N*`]_long_long` (*offset_expr*)	The long long integer *trace event argument*.

Table 16-1. NightTrace Functions

Syntax	Return Type
`blk_arg` (*byte_offset*[, *PR*]) `start_blk_arg` (*byte_offset*[, *PR*]) `end_blk_arg` (*byte_offset*[, *PR*]) `offset_blk_arg` (*byte_offset* , *offset_expr*)	The integer *trace event argument* at a particular byte offset in the argument space.

`blk_arg_bits`
(*byte_offset* , *bit_offset* , *bit_size*[, *PR*])
`start_blk_arg_bits`
(*byte_offset* , *bit_offset* , *bit_size*[, *PR*])
`end_blk_arg_bits`
(*byte_offset* , *bit_offset* , *bit_size*[, *PR*])
`offset_blk_arg_bits`
(*byte_offset* , *bit_offset* , *bit_size* , *offset_expr*)
	The integer *trace event argument* extracted as a signed bit field with a particular byte offset, bit offset, and bit size in the argument space.

`blk_arg_char` (*byte_offset*[, *PR*]) `start_blk_arg_char` (*byte_offset*[, *PR*]) `end_blk_arg_char` (*byte_offset*[, *PR*]) `offset_blk_arg_char` (*byte_offset* , *offset_expr*)	The signed character *trace event argument* at a particular byte offset in the argument space.
`blk_arg_dbl` (*byte_offset*[, *PR*]) `start_blk_arg_dbl` (*byte_offset*[, *PR*]) `end_blk_arg_dbl` (*byte_offset*[, *PR*]) `offset_blk_arg_dbl` (*byte_offset* , *offset_expr*)	The double-precision *trace event argument* at a particular byte offset in the argument space.
`blk_arg_flt` (*byte_offset*[, *PR*]) `start_blk_arg_flt` (*byte_offset*[, *PR*]) `end_blk_arg_flt` (*byte_offset*[, *PR*]) `offset_blk_arg_flt` (*byte_offset* , *offset_expr*)	The single-precision *trace event argument* at a particular byte offset in the argument space.
`blk_arg_long` (*byte_offset*[, *PR*]) `start_blk_arg_long` (*byte_offset*[, *PR*]) `end_blk_arg_long` (*byte_offset*[, *PR*]) `offset_blk_arg_long` (*byte_offset* , *offset_expr*)	The long integer *trace event argument* at a particular byte offset in the argument space.
`blk_arg_long_bits` (*byte_offset*[, *PR*]) `start_blk_arg_long_bits` (*byte_offset*[, *PR*]) `end_blk_arg_long_bits` (*byte_offset*[, *PR*]) `offset_blk_arg_long_bits` (*byte_offset* , *offset_expr*)	The long integer *trace event argument* extracted as a signed bit field with a particular byte offset, bit offset, and bit size in the argument space.
`blk_arg_long_dbl` (*byte_offset*[, *PR*]) `start_blk_arg_long_dbl` (*byte_offset*[, *PR*]) `end_blk_arg_long_dbl` (*byte_offset*[, *PR*]) `offset_blk_arg_long_dbl` (*byte_offset* , *offset_expr*)	The long double-precision *trace event argument* at a particular byte offset in the argument space.
`blk_arg_long_long` (*byte_offset*[, *PR*]) `start_blk_arg_long_long` (*byte_offset*[, *PR*]) `end_blk_arg_long_long` (*byte_offset*[, *PR*]) `offset_blk_arg_long_long` (*byte_offset* , *offset_expr*)	The long long integer *trace event argument* at a particular byte offset in the argument space.
`blk_arg_long_ubits` (*byte_offset*[, *PR*]) `start_blk_arg_long_ubits` (*byte_offset*[, *PR*]) `end_blk_arg_long_ubits` (*byte_offset*[, *PR*]) `offset_blk_arg_long_ubits` (*byte_offset* , *offset_expr*)	The long integer *trace event argument* extracted as an unsigned bit field with a particular byte offset, bit offset, and bit size in the argument space.

Table 16-1. NightTrace Functions

Syntax	Return Type
`blk_arg_short` (*byte_offset*[, *PR*]) `start_blk_arg_short` (*byte_offset*[, *PR*]) `end_blk_arg_short` (*byte_offset*[, *PR*]) `offset_blk_arg_short` (*byte_offset* , *offset_expr*)	The short integer *trace event argument* at a particular byte offset in the argument space.
`blk_arg_string` (*byte_offset* , *max_size*[, *PR*]) `start_blk_arg_string` (*byte_offset* , *max_size*[, *PR*]) `end_blk_arg_string` (*byte_offset* , *max_size*[, *PR*]) `offset_blk_arg_string` (*byte_offset* , *max_size* , *offset_expr*)	The null-byte terminated string *trace event argument* at a particular byte offset in the argument space.

`blk_arg_ubits`
(*byte_offset* , *bit_offset* , *bit_size*[, *PR*])
`start_blk_arg_ubits`
(*byte_offset* , *bit_offset* , *bit_size*[, *PR*])
`end_blk_arg_ubits`
(*byte_offset* , *bit_offset* , *bit_size*[, *PR*])
`offset_blk_arg_ubits`
(*byte_offset* , *bit_offset* , *bit_size* , *offset_expr*)
	The integer *trace event argument* extracted as an unsigned bit field with a particular byte offset, bit offset, and bit size in the argument space.

`blk_arg_uchar` (*byte_offset*[, *PR*]) `start_blk_arg_uchar` (*byte_offset*[, *PR*]) `end_blk_arg_uchar` (*byte_offset*[, *PR*]) `offset_blk_arg_uchar` (*byte_offset* , *offset_expr*)	The unsigned character *trace event argument* at a particular byte offset in the argument space.
`blk_arg_uint` (*byte_offset*[, *PR*]) `start_blk_arg_uint` (*byte_offset*[, *PR*]) `end_blk_arg_uint` (*byte_offset*[, *PR*]) `offset_blk_arg_uint` (*byte_offset*[, *PR*])	The unsigned integer *trace event argument* at a particular byte offset in the argument space, converted to type long.
`blk_arg_ulong_long` (*byte_offset*[, *PR*]) `start_blk_arg_ulong_long` (*byte_offset*[, *PR*]) `end_blk_arg_ulong_long` (*byte_offset*[, *PR*]) `offset_blk_arg_ulong_long` (*byte_offset*[, *PR*])	The unsigned long long integer *trace event argument* at a particular byte offset in the argument space.
`blk_arg_ushort` (*byte_offset*[, *PR*]) `start_blk_arg_ushort` (*byte_offset*[, *PR*]) `end_blk_arg_ushort` (*byte_offset*[, *PR*]) `offset_blk_arg_ushort` (*byte_offset* , *offset_expr*)	The unsigned short integer *trace event argument* at a particular byte offset in the argument space.
`num_args` [([*PR*])] `start_num_args` [([*PR*])] `end_num_args` [([*PR*])] `offset_num_args` (*offset_expr*)	The number of arguments associated with a *trace event*.
`pid` [([*PR*])] `start_pid` [([*PR*])] `end_pid` [([*PR*])] `offset_pid` (*offset_expr*)	The integer global process identifier (*PID*) associated with a *trace event*.
`thread_id` [([*PR*])] `start_thread_id` [([*PR*])] `end_thread_id` [([*PR*])] `offset_thread_id` (*offset_expr*)	The integer *thread* identifier (*thread ID*) associated with a *trace event*.

Table 16-1. NightTrace Functions

Syntax	Return Type
`task_id` [([*PR*])] `start_task_id` [([*PR*])] `end_task_id` [([*PR*])] `offset_task_id` (*offset_expr*)	The integer Ada task identifier associated with a *trace event*.
`tid` [([*PR*])] `start_tid` [([*PR*])] `end_tid` [([*PR*])] `offset_tid` (*offset_expr*)	The integer NightTrace thread identifier (*TID*) associated with a *trace event*.
`cpu` [([*PR*])] `start_cpu` [([*PR*])] `end_cpu` [([*PR*])] `offset_cpu` (*offset_expr*)	The integer logical CPU number associated with a *trace event*. This function is only valid when applied to events from NightTrace kernel trace event files.
`time` [([*PR*])] `start_time` [([*PR*])] `end_time` [([*PR*])] `offset_time` (*offset_expr*)	The double-precision floating point time, expressed in units of seconds, between a *trace event* and the earliest trace event from all *trace event files* currently in use.
`node_id` [([*PR*])] `start_node_id` [([*PR*])] `end_node_id` [([*PR*])] `offset_node_id` (*offset_expr*)	The internally-assigned integer *node identifier* associated with a *trace event*.
`pid_table_name` [([*PR*])] `start_pid_table_name` [([*PR*])] `end_pid_table_name` [([*PR*])] `offset_pid_table_name` (*offset_expr*)	The string describing the name of the process identifier table (*PID table*) associated with a *trace event*.
`tid_table_name` [([*PR*])] `start_tid_table_name` [([*PR*])] `end_tid_table_name` [([*PR*])] `offset_tid_table_name` (*offset_expr*)	The string describing the name of the internally-assigned thread identifier table (*TID table*) associated with a *trace event*.
`node_name` [([*PR*])] `start_node_name` [([*PR*])] `end_node_name` [([*PR*])] `offset_node_name` (*offset_expr*)	The string describing the name of the system from which a *trace event* was logged.
`process_name` [([*PR*])] `offset_process_name` (*offset_expr*)	The string describing the name of the process (*PID*) associated with a *trace event*.
`task_name` [([*PR*])] `offset_task_name` (*offset_expr*)	The string describing the name of the Ada *task* associated with a *trace event*.
`thread_name` [([*PR*])] `offset_thread_name` (*offset_expr*)	The string describing the name of the C *thread* associated with a *trace event*.
`event_gap` [([*PR*])] `state_gap` [([*PR*])]	The double-precision floating point time, expressed in units of seconds, between the instances of either a *trace event* or a *state*.

Table 16-1. NightTrace Functions

Syntax	Return Type
`state_dur` [([*PR*])]	The double-precision floating point time, expressed in units of seconds, of an instance of a *state*.
`event_matches` [([*PR*])] `state_matches` [([*PR*])] `summary_matches` [()]	The integer number of instances of either a *trace event* or a *state*.
`state_status` [([*PR*])]	The boolean status of a *state*; true if the *current time line* is within an instance of the state, false otherwise. See "state_status()" on page 16-137 for important details.
`offset` [([*PR*])] `start_offset` [([*PR*])] `end_offset` [([*PR*])]	The integer ordinal number (*offset*) of a *trace event*.
`min_offset` (*expr*) `max_offset` (*expr*)	The integer ordinal number (*offset*) of a *trace event* associated with a minimum or maximum occurrence of *expr.*
`min` (*expr*) `max` (*expr*) `avg` (*expr*) `sum` (*expr*)	The minimum, maximum, average, or sum of *expr* values before the *current time*. The return type is that of *expr.*
`get_string` (*table_name*[, *int_expr*])	The character string associated with item *int_expr* in string table *table_name*.
`get_item` (*table_name*, "*str_const*")	The first integer item number associated with string *str_const* in string table *table_name.*
`get_format` (*table_name*[, *int_expr*])	The character string associated with item *int_expr* in format table *table_name.*
`format` ("*format_string*" [, *arg*] ...)	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,

`arg2`

No parentheses are required

`arg2()`

No parentheses are required

`arg2(Myprof)`

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,

`arg1()`

Operates on the *current trace event*

`arg1(my_cond)`

Operates on the *profile reference* `my_cond`

`end_arg1()`

Operates on the *last completed instance* 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 state defined by the *profile reference* `my_state`

This manual uses the following conventions for function *parameters*:

PR

A user-defined *profile reference*. If supplied, the function applies to the specified profile. For more information, see "Profile Definition Panel" on page 13-1.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

expr

Any valid NightTrace *expression* (see "Overview" on page 16-1).

table_name

An unquoted character string that represents the name of a *string table* or *format table*.

int_expr

An integer expression that acts as an index into the specified *string table* or *format table*. *int_expr* must either match an identifying integer value in the *table_name* table, or the *table_name* table must have a `default item` line.

str_const

A string constant literal that acts as an index into the specified *string table*.

format_string

A character string that contains literal characters and conversion specifications. Conversion specifications modify zero or more *args*.

arg

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 *profile references*.

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:

current instance

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*.

last completed instance

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.

most recent instance

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.

Figure 16-1 illustrates some of these concepts with a State Graph.

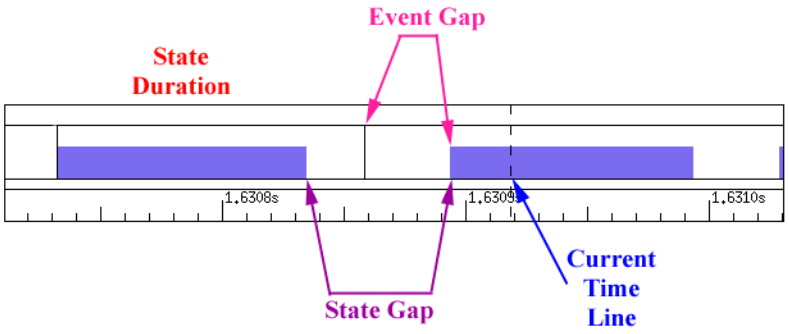


Figure 16-1. Function Terminology Illustrated

A more detailed example is illustrated in Figure 16-2.

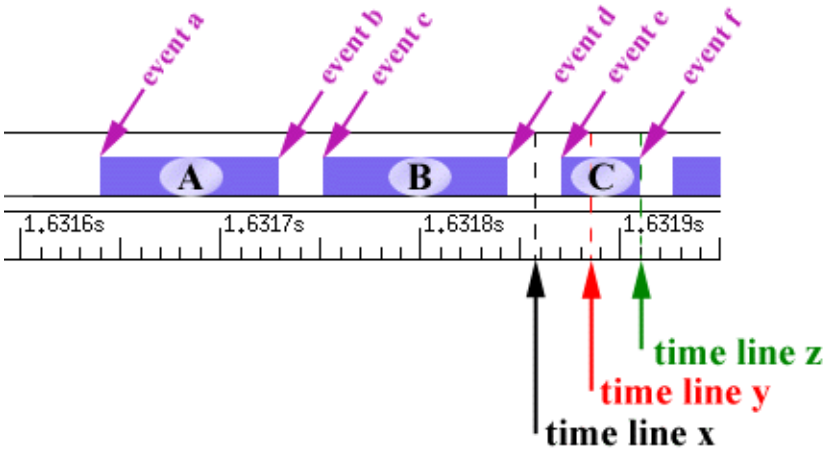


Figure 16-2. States and Events

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 16-2, the various "instances" would be defined as:

current instance

No current instance is defined since the current time line is not positioned within any instance of a state.

last completed instance

Instance B

most recent instance

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 16-2.

.

`state_status()`	false	The current time line was not positioned within a current instance of a state.
`state_gap()`	~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).
`state_dur()`	~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).
`state_matches()`	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).
`start_time()`	~1.631750	The time associated with event c. The function operated on the most recent instance of the state (instance B).
`end_time()`	~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 16-2, the various "instances" would be defined as:

current instance

Instance C

last completed instance

Instance B

most recent instance

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 16-2.

`state_status()`	true	The current time line was positioned inside a current instance of the state (instance C).
`state_gap()`	~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).
`state_dur()`	~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).
`state_matches()`	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).
`start_time()`	~1.631870	The time associated with event e. The function operated on the most recent instance of the state (instance C).
`end_time()`	~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 16-2, the various "instances" would be defined as:

current instance

No current instance is defined since the current time line is positioned on the *end event* of an instance of a state.

last completed instance

Instance C

most recent instance

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 16-2.

.

`state_status()`	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 *end event* of the state (event f), the corresponding instance is said to have already completed.
`state_gap()`	~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).
`state_dur()`	~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).
`state_matches()`	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).
`start_time()`	~1.631870	The time associated with event e. The function operated on the most recent instance of the state (instance C).
`end_time()`	~1.631910	The time associated with event f. The function operated on the last completed instance of the state (instance C).

String Functions

The string functions compare two strings. They include the following:

- `strcmp()`
- `strncmp()`

strcmp()

DESCRIPTION

The `strcmp()` function compares the two strings, *s1* and *s2*. It returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*.

SYNTAX

```
strcmp(s1, s2);
```

PARAMETERS

s1

The string to be compared to *s2*

s2

The string to be compared to *s1*

RETURN TYPE

integer

SEE ALSO

- "strncmp()" on page 16-17

strncmp()

DESCRIPTION

The `strncmp()` function is similar to `strcmp()` in that it compares two strings, *s1* and *s2*, and returns an integer less than, equal to, or greater than zero if *s1* is found, respectively, to be less than, to match, or be greater than *s2*. However, `strncmp()` only compares the first (at most) *n* bytes of *s1* and *s2*.

SYNTAX

```
strncmp(s1, s2) n;
```

PARAMETERS

s1

The string to be compared to *s2*

s2

The string to be compared to *s1*

n

The maximum number of bytes in *s1* and *s2* to be compared

RETURN TYPE

integer

SEE ALSO

- “strcmp()” on page 16-16

Trace Event Functions

The trace event functions operate on either the *profile reference* specified to that function or the *current trace event*. They include the following:

- `id`
- `arg`
- `arg_dbl()`
- `arg_long()`
- `arg_long_dbl()`
- `arg_long_long()`
- `blk_arg()`
- `blk_arg_bits()`
- `blk_arg_char()`
- `blk_arg_dbl()`
- `blk_arg_flt()`
- `blk_arg_long()`
- `blk_arg_long_bits()`
- `blk_arg_long_dbl()`
- `blk_arg_long_long()`
- `blk_arg_long_ubits()`
- `blk_arg_short()`
- `blk_arg_string()`
- `blk_arg_ubits()`
- `blk_arg_uchar()`
- `blk_arg_uint()`
- `blk_arg_ulong_long()`
- `blk_arg_ushort()`
- `num_args()`
- `pid()`
- `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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function returns the *trace event ID* of the last instance of the trace event which satisfies the conditions of the specified specified profile. If omitted, the function returns the *trace event ID* of the current trace event. For more information, see "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "start_id()" on page 16-62
- "end_id()" on page 16-99
- "offset_id()" on page 16-140

arg()

DESCRIPTION

The `arg()` function returns the value of a particular *trace event argument*.

SYNTAX

`arg`[*N*] [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *trace event*. Defaults to 1.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “arg_long()” on page 16-23
- “arg_dbl()” on page 16-22
- “arg_long_long()” on page 16-25
- “arg_long_dbl()” on page 16-24
- “num_args()” on page 16-43
- “start_arg()” on page 16-63
- “end_arg()” on page 16-100
- “offset_arg()” on page 16-141

arg_dbl()

DESCRIPTION

The `arg_dbl()` function returns the value of a particular *trace event argument*.

SYNTAX

`arg`[*N*]`_dbl` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *trace event*. Defaults to 1.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- "arg()" on page 16-21
- "arg_long()" on page 16-23
- "num_args()" on page 16-43
- "start_arg_dbl()" on page 16-64
- "end_arg_dbl()" on page 16-101
- "offset_arg_dbl()" on page 16-142

arg_long()

DESCRIPTION

The `arg_long()` function returns the value of a particular *trace event argument*.

SYNTAX

`arg`[*N*]`_long` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *trace event*. Defaults to 1.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- “arg()” on page 16-21
- “num_args()” on page 16-43
- “start_arg_long()” on page 16-65
- “end_arg_long()” on page 16-102
- “offset_arg_long()” on page 16-143

arg_long_dbl()

DESCRIPTION

The `arg_long_dbl()` function returns the value of a particular *trace event argument*.

SYNTAX

`arg`[*N*]`_long_dbl` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *trace event*. Defaults to 1.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

long double-precision floating point

SEE ALSO

- "arg()" on page 16-21
- "num_args()" on page 16-43
- "start_arg_long_dbl()" on page 16-66
- "end_arg_long_dbl()" on page 16-103
- "offset_arg_long_dbl()" on page 16-144

arg_long_long()

DESCRIPTION

The `arg_long_long()` function returns the value of a particular *trace event argument*.

SYNTAX

`arg`[*N*]`_long_long` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *trace event*. Defaults to 1.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

long long integer

SEE ALSO

- “arg()” on page 16-21
- “num_args()” on page 16-43
- “start_arg_long_long()” on page 16-67
- “end_arg_long_long()” on page 16-104
- “offset_arg_long_long()” on page 16-145

blk_arg()

DESCRIPTION

The `blk_arg()` function returns the value of a trace event argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "start_blk_arg()" on page 16-68
- "end_blk_arg()" on page 16-105
- "offset_blk_arg()" on page 16-146

blk_arg_bits()

DESCRIPTION

The `blk_arg_bits()` function returns the value of a trace event signed bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with an event.

SYNTAX

`blk_arg_bits` (*byte_offset* , *bit_offset* , *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “num_args()” on page 16-43
- “start_blk_arg_bits()” on page 16-69
- “end_blk_arg_bits()” on page 16-106
- “offset_blk_arg_bits()” on page 16-147

blk_arg_char()

DESCRIPTION

The `blk_arg_char()` function returns the value of a trace event signed character argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg_char` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "start_blk_arg_char()" on page 16-70
- "end_blk_arg_char()" on page 16-107
- "offset_blk_arg_char()" on page 16-148

blk_arg_dbl()

DESCRIPTION

The `blk_arg_dbl()` function returns the value of a trace event double-precision floating point argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg_dbl` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- “num_args()” on page 16-43
- “start_blk_arg_dbl()” on page 16-71
- “end_blk_arg_dbl()” on page 16-108
- “offset_blk_arg_dbl()” on page 16-149

blk_arg_flt()

DESCRIPTION

The `blk_arg_flt()` function returns the value of a trace event single-precision floating point argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg_flt` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- "num_args()" on page 16-43
- "start_blk_arg_flt()" on page 16-72
- "end_blk_arg_flt()" on page 16-109
- "offset_blk_arg_flt()" on page 16-150

blk_arg_long()

DESCRIPTION

The `blk_arg_long()` function returns the value of a trace event long integer argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg_long` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

long integer

SEE ALSO

- “num_args()” on page 16-43
- “start_blk_arg_long()” on page 16-73
- “end_blk_arg_long()” on page 16-110
- “offset_blk_arg_long()” on page 16-151

blk_arg_long_bits()

DESCRIPTION

The `blk_arg_long_bits()` function returns the value of a trace event signed long bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with an event.

SYNTAX

`blk_arg_long_bits` (*byte_offset*, *bit_offset*, *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

long integer

SEE ALSO

- "num_args()" on page 16-43
- "start_blk_arg_long_bits()" on page 16-74
- "end_blk_arg_long_bits()" on page 16-111
- "offset_blk_arg_long_bits()" on page 16-152

blk_arg_long_dbl()

DESCRIPTION

The `blk_arg_long_dbl()` function returns the value of a trace event long double-precision floating point argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg_long_dbl` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

long double-precision floating point

SEE ALSO

- “num_args()” on page 16-43
- “start_blk_arg_long_dbl()” on page 16-75
- “end_blk_arg_long_dbl()” on page 16-112
- “offset_blk_arg_long_dbl()” on page 16-153

blk_arg_long_long()

DESCRIPTION

The `blk_arg_long_long()` function returns the value of a trace event long long integer argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg_long_long` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

long long integer

SEE ALSO

- "num_args()" on page 16-43
- "start_blk_arg_long_long()" on page 16-76
- "end_blk_arg_long_long()" on page 16-113
- "offset_blk_arg_long_long()" on page 16-154

blk_arg_long_ubits()

DESCRIPTION

The `blk_arg_long_ubits()` function returns the value of a trace event unsigned long integer bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with an event.

SYNTAX

`blk_arg_long_ubits` (*byte_offset* , *bit_offset* , *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

long long integer

SEE ALSO

- “num_args()” on page 16-43
- “start_blk_arg_long_ubits()” on page 16-77
- “end_blk_arg_long_ubits()” on page 16-114
- “offset_blk_arg_long_ubits()” on page 16-155

blk_arg_short()

DESCRIPTION

The `blk_arg_short()` function returns the value of a trace event short integer argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg_short` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "start_blk_arg_short()" on page 16-78
- "end_blk_arg_short()" on page 16-115
- "offset_blk_arg_short()" on page 16-156

blk_arg_string()

DESCRIPTION

The `blk_arg_string()` function returns the value of a trace event null terminated string argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg_string` (*byte_offset* , *max_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk* or *trace_event_string*.

max_size

Specifies the maximum length of string that might be returned. If the arguments were recorded with *trace_event_blk*, this is also the total number of bytes allocated in the block for the string, regardless of its actual lenght.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

string

SEE ALSO

- “num_args()” on page 16-43
- “start_blk_arg_string()” on page 16-79
- “end_blk_arg_string()” on page 16-116
- “offset_blk_arg_string()” on page 16-157

blk_arg_ubits()

DESCRIPTION

The `blk_arg_ubits()` function returns the value of a trace event unsigned bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with an event.

SYNTAX

`blk_arg_ubits` (*byte_offset* , *bit_offset* , *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "start_blk_arg_ubits()" on page 16-80
- "end_blk_arg_ubits()" on page 16-117
- "offset_blk_arg_ubits()" on page 16-158

blk_arg_uchar()

DESCRIPTION

The `blk_arg_uchar()` function returns the value of a trace event unsigned character argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg_uchar` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “num_args()” on page 16-43
- “start_blk_arg_uchar()” on page 16-81
- “end_blk_arg_uchar()” on page 16-118
- “offset_blk_arg_uchar()” on page 16-159

blk_arg_uint()

DESCRIPTION

The `blk_arg_uint()` function converts the unsigned integer trace event argument at a particular byte offset in the argument space to a long.

NOTE

You can convert the long return value to an unsigned value using the cast operator. For example:

```
(unsigned long) blk_arg_uint(0) > 0x80000000
```

SYNTAX

`blk_arg_uint` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

long

SEE ALSO

- "num_args()" on page 16-43
- "start_blk_arg_uint()" on page 16-82
- "end_blk_arg_uint()" on page 16-119
- "offset_blk_arg_uint()" on page 16-160

blk_arg_ulong_long()

DESCRIPTION

The `blk_arg_ulong_long()` function returns the value of a trace event unsigned long long integer argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg_ulong_long` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

unsigned long long integer

SEE ALSO

- “num_args()” on page 16-43
- “start_blk_arg_ulong_long()” on page 16-83
- “end_blk_arg_ulong_long()” on page 16-120
- “offset_blk_arg_ulong_long()” on page 16-161

blk_arg_ushort()

DESCRIPTION

The `blk_arg_ushort()` function returns the value of a trace event unsigned short integer argument located at a particular byte offset in the argument space associated with an event.

SYNTAX

`blk_arg_ushort` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the specified argument for the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "start_blk_arg_ushort()" on page 16-84
- "end_blk_arg_ushort()" on page 16-121
- "offset_blk_arg_ushort()" on page 16-162

num_args()

DESCRIPTION

The `num_args()` function returns the number of arguments logged with a *trace event*. For events recorded with `trace_event_blk()`, it returns the number of bytes recorded in the argument space.

SYNTAX

`num_args` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function returns the number of arguments of the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the number of arguments of the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “arg()” on page 16-21
- “start_num_args()” on page 16-85
- “end_num_args()” on page 16-122
- “offset_num_args()” on page 16-163

pid()

DESCRIPTION

The `pid()` function returns the global process identifier (*PID*) associated with a *trace event*.

NOTE

All Linux threads within the same program share the same PID value. For trace events generated with the NightTrace Logging API, the value logged as the process identifier is the common PID. For kernel events, the value logged for the process identifier is the actually the thread's TID (see **`gettid(2)`**).

SYNTAX

`pid` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function returns the global process identifier of the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the global process identifier of the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "start_pid()" on page 16-86
- "end_pid()" on page 16-123
- "offset_pid()" on page 16-164

thread_id()

DESCRIPTION

The `thread_id()` function returns the *thread* identifier associated with a *trace event*. The thread identifier is the value of the system call **`gettid(2)`**.

SYNTAX

`thread_id` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function returns the thread identifier of the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the thread identifier of the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “start_thread_id()” on page 16-87
- “end_thread_id()” on page 16-124
- “offset_thread_id()” on page 16-165

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function returns the Ada task identifier of the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the Ada task identifier of the *current trace event*. For more information, see "Profile References" on page 16-193Profile References.

RETURN TYPE

integer

SEE ALSO

- "start_task_id()" on page 16-88
- "end_task_id()" on page 16-125
- "offset_task_id()" on page 16-166

tid()

DESCRIPTION

The `tid()` function returns the internally-assigned NightTrace thread identifier (*TID*) associated with a *trace event*.

SYNTAX

`tid` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function returns the NightTrace thread identifier of the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the NightTrace thread identifier of the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “start_tid()” on page 16-89
- “end_tid()” on page 16-126
- “offset_tid()” on page 16-167

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 NightTrace kernel trace event files. Kernel tracing is not supported on all operating system distributions. See "Kernel Dependencies" on page B-1 for more information.

SYNTAX

`cpu` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function returns the logical CPU number of the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the logical CPU number of the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "start_cpu()" on page 16-90
- "end_cpu()" on page 16-127
- "offset_cpu()" on page 16-168

offset()

DESCRIPTION

The `offset()` function returns the ordinal number (*offset*) of a *trace event*.

SYNTAX

`offset` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function returns the ordinal number (*offset*) of the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the ordinal number (*offset*) of the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “start_offset()” on page 16-91
- “end_offset()” on page 16-128
- “min_offset()” on page 16-181
- “max_offset()” on page 16-182

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function returns the time, in seconds, of the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the time, in seconds, of the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- "event_gap()" on page 16-58
- "start_time()" on page 16-92
- "end_time()" on page 16-129
- "state_gap()" on page 16-134
- "state_dur()" on page 16-135
- "offset_time()" on page 16-169

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 16-54 for more information about this function.)

SYNTAX

`node_id` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function returns the node identifier of the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the node identifier of the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "start_node_id()" on page 16-93
- "offset_node_id()" on page 16-170
- "end_node_id()" on page 16-130

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 profile. If omitted, the function returns the name of the process identifier table (*PID table*) of the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

string

SEE ALSO

- "start_pid_table_name()" on page 16-94
- "offset_pid_table_name()" on page 16-171
- "end_pid_table_name()" on page 16-131

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 profile. If omitted, the function returns the name of the thread identifier table (*TID table*) of the *current trace event.* For more information, see “Profile References” on page 16-193.

RETURN TYPE

string

SEE ALSO

- “start_tid_table_name()” on page 16-95
- “offset_tid_table_name()” on page 16-172
- “end_tid_table_name()” on page 16-132

node_name()

DESCRIPTION

The `node_name()` function returns the name of the system from which a *trace event* was logged.

SYNTAX

`node_name` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 profile was logged. If omitted, the function returns the name of the system from which the *current trace event* was logged. For more information, see "Profile References" on page 16-193.

RETURN TYPE

string

SEE ALSO

- "start_node_name()" on page 16-96
- "offset_node_name()" on page 16-173
- "end_node_name()" on page 16-133

process_name()

DESCRIPTION

The `process_name()` function returns the name of the process associated with a *trace event*.

SYNTAX

`process_name` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 profile. If omitted, the function returns the name associated with the *PID* of the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

string

SEE ALSO

- “offset_process_name()” on page 16-174

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 profile. If omitted, the function returns the name of the task associated with the *current trace event*. For more information, see "Profile References" on page 16-193.

RETURN TYPE

string

SEE ALSO

- "offset_task_name()" on page 16-175

thread_name()

DESCRIPTION

The `thread_name()` function returns the thread name associated with a *trace event*.

Thread names are only available when user trace data is loaded and then only for threads registered with the NightTrace Logging API.

See “Threads and Logging” on page 2-33 for a discussion of the threads and the NightTrace Logging API.

SYNTAX

`thread_name` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function returns the thread name associated with the last instance of the trace event which satisfies the conditions for the specified profile. If omitted, the function returns the thread name associated with the *current trace event*. For more information, see “Profile References” on page 16-193.

RETURN TYPE

string

SEE ALSO

- “offset_thread_name()” on page 16-176

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function calculates the gap between the two most recent occurrences of events which satisfy the conditions of the specified profile. If omitted, the function calculates the gap between the current trace event and the event immediately preceding it. For more information, see "Profile References" on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- "time()" on page 16-50
- "state_gap()" on page 16-134
- "state_dur()" on page 16-135

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. If supplied, the function calculates the number of occurrences of events which satisfy the conditions of the specified profile 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “summary_matches()” on page 16-183

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" on page 16-60
- "End Functions" on page 16-97
- "Multi-State Functions" on page 16-134

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 *profile reference* specified to the function, or the state being currently defined. Thus, if a profile 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 16-97) provide information about the *last completed instance of a state*.

Start functions include the following:

- `start_id()`
- `start_arg()`
- `start_arg_dbl()`
- `start_arg_long()`
- `start_arg_long()`

- `start_arg_long_dbl()`
- `start_arg_long_long()`
- `start_blk_arg()`
- `start_blk_arg_bits()`
- `start_blk_arg_char()`
- `start_blk_arg_dbl()`
- `start_blk_arg_flt()`
- `start_blk_arg_long()`
- `start_blk_arg_long_bits()`
- `start_blk_arg_long_dbl()`
- `start_blk_arg_long_long()`
- `start_blk_arg_long_ubits()`
- `start_blk_arg_short()`
- `start_blk_arg_string()`
- `start_blk_arg_ubits()`
- `start_blk_arg_uchar()`
- `start_blk_arg_uint()`
- `start_blk_arg_ulong_long()`
- `start_blk_arg_ushort()`
- `start_num_args()`
- `start_pid()`
- `start_thread_id()`
- `start_task_id()`
- `start_tid()`
- `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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "id()" on page 16-20
- "end_id()" on page 16-99
- "offset_id()" on page 16-140

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*] [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *start event*. Defaults to 1.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “arg()” on page 16-21
- “start_arg_dbl()” on page 16-64
- “start_num_args()” on page 16-85
- “end_arg()” on page 16-100
- “offset_arg()” on page 16-141

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` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *start event*. Defaults to 1.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- "arg_dbl()" on page 16-22
- "start_arg()" on page 16-63
- "start_num_args()" on page 16-85
- "end_arg_dbl()" on page 16-101
- "offset_arg_dbl()" on page 16-142

start_arg_long()

DESCRIPTION

The `start_arg_long()` 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*]`_long` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *start event*. Defaults to 1.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- "arg_dbl()" on page 16-22
- "start_arg()" on page 16-63
- "start_num_args()" on page 16-85
- "end_arg_dbl()" on page 16-101
- "offset_arg_long()" on page 16-143

start_arg_long_dbl()

DESCRIPTION

The `start_arg_long_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*]`_long_dbl` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *trace event*. Defaults to 1.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

long double-precision floating point

SEE ALSO

- "num_args()" on page 16-43
- "arg_long_dbl()" on page 16-24
- "end_arg_long_dbl()" on page 16-103
- "offset_arg_long_dbl()" on page 16-144

start_arg_long_long()

DESCRIPTION

The `start_arg_long_long()` 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*]`_long_long` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *trace event*. Defaults to 1.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

long long integer

SEE ALSO

- “arg_long_long()” on page 16-25
- “num_args()” on page 16-43
- “end_arg_long_long()” on page 16-104
- “offset_arg_long_long()” on page 16-145

start_blk_arg()

DESCRIPTION

The `start_blk_arg()` function returns the value of a trace event argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg()" on page 16-26
- "end_blk_arg()" on page 16-105
- "offset_blk_arg()" on page 16-146

start_blk_arg_bits()

DESCRIPTION

The `start_blk_arg_bits()` function returns the value of a trace event signed bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_bits` (*byte_offset* , *bit_offset* , *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_bits()” on page 16-27
- “end_blk_arg_bits()” on page 16-106
- “offset_blk_arg_bits()” on page 16-147

start_blk_arg_char()

DESCRIPTION

The `start_blk_arg_char()` function returns the value of a trace event signed character argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_char` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_char()" on page 16-28
- "end_blk_arg_char()" on page 16-107
- "offset_blk_arg_char()" on page 16-148

start_blk_arg_dbl()

DESCRIPTION

The `start_blk_arg_dbl()` function returns the value of a trace event double-precision floating point argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_dbl` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_dbl()” on page 16-29
- “end_blk_arg_dbl()” on page 16-108
- “offset_blk_arg_dbl()” on page 16-149

start_blk_arg_flt()

DESCRIPTION

The `start_blk_arg_flt()` function returns the value of a trace event single-precision floating point argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start blk_arg_flt` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_flt()" on page 16-30
- "end_blk_arg_flt()" on page 16-109
- "offset_blk_arg_flt()" on page 16-150

start_blk_arg_long()

DESCRIPTION

The `start_blk_arg_long()` function returns the value of a trace event long integer argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_long` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

long integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_long()” on page 16-31
- “end_blk_arg_long()” on page 16-110
- “offset_blk_arg_long()” on page 16-151

start_blk_arg_long_bits()

DESCRIPTION

The `start_blk_arg_long_bits()` function returns the value of a trace event signed long bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_long_bits` (*byte_offset* , *bit_offset* , *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

long integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_long_bits()" on page 16-32
- "end_blk_arg_long_bits()" on page 16-111
- "offset_blk_arg_long_bits()" on page 16-152

start_blk_arg_long_dbl()

DESCRIPTION

The `start_blk_arg_long_dbl()` function returns the value of a trace event long double-precision floating point argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_long_dbl` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

long double-precision floating point

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_long_dbl()” on page 16-33
- “end_blk_arg_long_dbl()” on page 16-112
- “offset_blk_arg_long_dbl()” on page 16-153

start_blk_arg_long_long()

DESCRIPTION

The `start_blk_arg_long_long()` function returns the value of a trace event long long integer argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_long_long` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

long long integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_long_long()" on page 16-34
- "end_blk_arg_long_long()" on page 16-113
- "offset_blk_arg_long_long()" on page 16-154

start_blk_arg_long_ubits()

DESCRIPTION

The `start_blk_arg_long_ubits()` function returns the value of a trace event unsigned long integer bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_long_ubits` (*byte_offset* , *bit_offset* , *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

long long integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_long_ubits()” on page 16-35
- “end_blk_arg_long_ubits()” on page 16-114
- “offset_blk_arg_long_ubits()” on page 16-155

start_blk_arg_short()

DESCRIPTION

The `start_blk_arg_short()` function returns the value of a trace event short integer argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_short` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_short()" on page 16-36
- "end_blk_arg_short()" on page 16-115
- "offset_blk_arg_short()" on page 16-156

start_blk_arg_string()

DESCRIPTION

The `start_blk_arg_string()` function returns the value of a trace event null terminated string argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_string` (*byte_offset* , *max_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk* or *trace_event_string*.

max_size

Specifies the maximum length of string that might be returned. If the arguments were recorded with *trace_event_blk*, this is also the total number of bytes allocated in the block for the string, regardless of its actual lenght.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

string

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_string()” on page 16-37
- “end_blk_arg_string()” on page 16-116
- “offset_blk_arg_string()” on page 16-157

start_blk_arg_ubits()

DESCRIPTION

The `start_blk_arg_ubits()` function returns the value of a trace event unsigned bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_ubits` (*byte_offset* , *bit_offset* , *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_ubits()" on page 16-38
- "end_blk_arg_ubits()" on page 16-117
- "offset_blk_arg_ubits()" on page 16-158

start_blk_arg_uchar()

DESCRIPTION

The `start_blk_arg_uchar()` function returns the value of a trace event unsigned character argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_uchar` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_uchar()” on page 16-39
- “end_blk_arg_uchar()” on page 16-118
- “offset_blk_arg_uchar()” on page 16-159

start_blk_arg_uint()

DESCRIPTION

The `start_blk_arg_uint()` function converts the unsigned integer trace event argument at a particular byte offset in the argument space associated with the *start event* of the *most recent instance of a state* to a long.

NOTE

You can convert the long return value to an unsigned value using the cast operator. For example:

```
(unsigned long) start_blk_arg_uint(0) > 0x80000000
```

SYNTAX

`start_blk_arg_uint` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

unsigned integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_uint()" on page 16-40
- "end_blk_arg_uint()" on page 16-119
- "offset_blk_arg_uint()" on page 16-160

start_blk_arg_ulong_long()

DESCRIPTION

The `start_blk_arg_ulong_long()` function returns the value of a trace event unsigned long long integer argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_ulong_long` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

unsigned long long integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_ulong_long()” on page 16-41
- “end_blk_arg_ulong_long()” on page 16-120
- “offset_blk_arg_ulong_long()” on page 16-161

start_blk_arg_ushort()

DESCRIPTION

The `start_blk_arg_ushort()` function returns the value of a trace event unsigned short integer argument located at a particular byte offset in the argument space associated with the event associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_blk_arg_ushort` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_ushort()" on page 16-42
- "end_blk_arg_ushort()" on page 16-121
- "offset_blk_arg_ushort()" on page 16-162

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*. For events recorded with `trace_event_blk()`, it returns the number of bytes recorded in the argument space.

SYNTAX

`start_num_args` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “start_arg()” on page 16-63
- “num_args()” on page 16-43
- “end_num_args()” on page 16-122
- “offset_num_args()” on page 16-163

start_pid()

DESCRIPTION

The `start_pid()` function returns the PID associated with the *start event* of the *most recent instance of a state*.

NOTE

All Linux threads within the same program share the same PID value. For trace events generated with the NightTrace Logging API, the value logged as the process identifier is the common PID. For kernel events, the value logged for the process identifier is the actually the thread's TID (see **`gettid(2)`**).

SYNTAX

`start_pid` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "pid()" on page 16-44
- "end_pid()" on page 16-123
- "offset_pid()" on page 16-164

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*. The thread identifier is the value of the system call **`gettid(2)`**.

SYNTAX

`start_thread_id` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “thread_id()” on page 16-45
- “end_thread_id()” on page 16-124
- “offset_thread_id()” on page 16-165

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "task_id()" on page 16-46
- "end_task_id()" on page 16-125
- "offset_task_id()" on page 16-166

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “tid()” on page 16-47
- “end_tid()” on page 16-126
- “offset_tid()” on page 16-167

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 NightTrace kernel trace event files. Kernel tracing is not supported on all operating systems. See "Kernel Dependencies" on page B-1 for more information.

SYNTAX

`start_cpu` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "cpu()" on page 16-48
- "end_cpu()" on page 16-127
- "offset_cpu()" on page 16-168

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “offset()” on page 16-49
- “end_offset()” on page 16-128

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- "time()" on page 16-50
- "end_time()" on page 16-129
- "state_gap()" on page 16-134
- "state_dur()" on page 16-135
- "offset_time()" on page 16-169

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “node_id()” on page 16-51
- “offset_node_id()” on page 16-170
- “end_node_id()” on page 16-130

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

string

SEE ALSO

- "pid_table_name()" on page 16-52
- "offset_pid_table_name()" on page 16-171
- "end_pid_table_name()" on page 16-131

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

string

SEE ALSO

- “tid_table_name()” on page 16-53
- “offset_tid_table_name()” on page 16-172
- “end_tid_table_name()” on page 16-132

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

string

SEE ALSO

- "node_name()" on page 16-54
- "offset_node_name()" on page 16-173
- "end_node_name()" on page 16-133

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 *profile reference* specified to the function, or the state being currently defined. Thus, if a profile 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 16-60) provide information about the *most recent instance of a state*.

End functions include:

- `end_id()`
- `end_arg()`
- `end_arg_dbl()`
- `end_arg_long_dbl()`
- `end_arg_long_long()`
- `end_blk_arg()`
- `end_blk_arg_bits()`
- `end_blk_arg_char()`
- `end_blk_arg_dbl()`
- `end_blk_arg_flt()`
- `end_blk_arg_long()`
- `end_blk_arg_long_bits()`
- `end_blk_arg_long_dbl()`
- `end_blk_arg_long_long()`
- `end_blk_arg_long_ubits()`
- `end_blk_arg_short()`
- `end_blk_arg_string()`
- `end_blk_arg_ubits()`
- `end_blk_arg_uchar()`
- `end_blk_arg_uint()`
- `end_blk_arg_ulong_long()`

- `end_blk_arg_ushort()`
- `end_num_args()`
- `end_pid()`
- `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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “id()” on page 16-20
- “start_id()” on page 16-62
- “offset_id()” on page 16-140

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*] [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the trace event. Defaults to 1.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "arg()" on page 16-21
- "start_arg()" on page 16-63
- "end_arg()" on page 16-100
- "end_num_args()" on page 16-122
- "offset_arg()" on page 16-141

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` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the trace event. Defaults to 1.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- “arg_dbl()” on page 16-22
- “start_arg_dbl()” on page 16-64
- “end_num_args()” on page 16-122
- “offset_arg_dbl()” on page 16-142

end_arg_long()

DESCRIPTION

The `end_arg_long()` 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*]`_long` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the trace event. Defaults to 1.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- "arg_long()" on page 16-23
- "start_arg_long()" on page 16-65
- "end_num_args()" on page 16-122
- "offset_arg_long()" on page 16-143

end_arg_long_dbl()

DESCRIPTION

The `end_arg_long_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

`end_arg`[*N*]`_long_dbl` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *trace event*. Defaults to 1.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

long double-precision floating point

SEE ALSO

- “num_args()” on page 16-43
- “arg_long_dbl()” on page 16-24
- “start_arg_long_dbl()” on page 16-66
- “offset_arg_long_dbl()” on page 16-144

end_arg_long_long()

DESCRIPTION

The `end_arg_long_long()` function returns the value of a particular *trace event argument.*

SYNTAX

`end_arg`[*N*]`_long_long` [([*PR*])]

PARAMETERS

N

Specifies the *N*th argument logged with the *trace event*. Defaults to 1.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

long long integer

SEE ALSO

- "arg_long_long()" on page 16-25
- "num_args()" on page 16-43
- "start_arg_long_long()" on page 16-67
- "offset_arg_long_long()" on page 16-145

end_blk_arg()

DESCRIPTION

The `end_blk_arg()` function returns the value of a trace event argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg()” on page 16-26
- “start_blk_arg()” on page 16-68
- “offset_blk_arg()” on page 16-146

end_blk_arg_bits()

DESCRIPTION

The `end_blk_arg_bits()` function returns the value of a trace event signed bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_bits` (*byte_offset* , *bit_offset* , *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_bits()" on page 16-27
- "start_blk_arg_bits()" on page 16-69
- "offset_blk_arg_bits()" on page 16-147

end_blk_arg_char()

DESCRIPTION

The `end_blk_arg_char()` function returns the value of a trace event signed character argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_char` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_char()” on page 16-28
- “start_blk_arg_char()” on page 16-70
- “offset_blk_arg_char()” on page 16-148

end_blk_arg_dbl()

DESCRIPTION

The `end_blk_arg_dbl()` function returns the value of a trace event double-precision floating point argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_dbl` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_dbl()" on page 16-29
- "start_blk_arg_dbl()" on page 16-71
- "offset_blk_arg_dbl()" on page 16-149

end_blk_arg_flt()

DESCRIPTION

The `end_blk_arg_flt()` function returns the value of a trace event single-precision floating point argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_flt` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_flt()” on page 16-30
- “start_blk_arg_flt()” on page 16-72
- “offset_blk_arg_flt()” on page 16-150

end_blk_arg_long()

DESCRIPTION

The `end_blk_arg_long()` function returns the value of a trace event long integer argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_long` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

long integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_long()" on page 16-31
- "start_blk_arg_long()" on page 16-73
- "offset_blk_arg_long()" on page 16-151

end_blk_arg_long_bits()

DESCRIPTION

The `end_blk_arg_long_bits()` function returns the value of a trace event signed long bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_long_bits` (*byte_offset* , *bit_offset* , *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

long integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_long_bits()” on page 16-32
- “start_blk_arg_long_bits()” on page 16-74
- “offset_blk_arg_long_bits()” on page 16-152

end_blk_arg_long_dbl()

DESCRIPTION

The `end_blk_arg_long_dbl()` function returns the value of a trace event long double-precision floating point argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_long_dbl` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

long double-precision floating point

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_long_dbl()" on page 16-33
- "start_blk_arg_long_dbl()" on page 16-75
- "offset_blk_arg_long_dbl()" on page 16-153

end_blk_arg_long_long()

DESCRIPTION

The `end_blk_arg_long_long()` function returns the value of a trace event long long integer argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_long_long` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

long long integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_long_long()” on page 16-34
- “start_blk_arg_long_long()” on page 16-76
- “offset_blk_arg_long_long()” on page 16-154

end_blk_arg_long_ubits()

DESCRIPTION

The `end_blk_arg_long_ubits()` function returns the value of a trace event unsigned long integer bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_long_ubits` (*byte_offset* , *bit_offset* , *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

long long integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_long_ubits()" on page 16-35
- "start_blk_arg_long_ubits()" on page 16-77
- "offset_blk_arg_long_ubits()" on page 16-155

end_blk_arg_short()

DESCRIPTION

The `end_blk_arg_short()` function returns the value of a trace event short integer argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_short` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_short()” on page 16-36
- “start_blk_arg_short()” on page 16-78
- “offset_blk_arg_short()” on page 16-156

end_blk_arg_string()

DESCRIPTION

The `end_blk_arg_string()` function returns the value of a trace event null terminated string argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_string` (*byte_offset* , *max_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk* or *trace_event_string*.

max_size

Specifies the maximum length of string that might be returned. If the arguments were recorded with *trace_event_blk*, this is also the total number of bytes allocated in the block for the string, regardless of its actual lenght.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

string

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_string()" on page 16-37
- "start_blk_arg_string()" on page 16-79
- "offset_blk_arg_string()" on page 16-157

end_blk_arg_ubits()

DESCRIPTION

The `end_blk_arg_ubits()` function returns the value of a trace event unsigned bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_ubits` (*byte_offset* , *bit_offset* , *bit_size*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_ubits()” on page 16-38
- “start_blk_arg_ubits()” on page 16-80
- “offset_blk_arg_ubits()” on page 16-158

end_blk_arg_uchar()

DESCRIPTION

The `end_blk_arg_uchar()` function returns the value of a trace event unsigned character argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_uchar` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_uchar()" on page 16-39
- "start_blk_arg_uchar()" on page 16-81
- "offset_blk_arg_uchar()" on page 16-159

end_blk_arg_uint()

DESCRIPTION

The `end_blk_arg_uint()` function converts the unsigned integer trace event argument at a particular byte offset in the argument space associated with the *end event* of the *most recent instance of a state* to a long.

NOTE

You can convert the long return value to an unsigned value using the cast operator. For example:

```
(unsigned long) end_blk_arg_uint(0) > 0x80000000
```

SYNTAX

`end_blk_arg_uint` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

unsigned integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_uint()” on page 16-40
- “start_blk_arg_uint()” on page 16-82
- “offset_blk_arg_uint()” on page 16-160

end_blk_arg_ulong_long()

DESCRIPTION

The `end_blk_arg_ulong_long()` function returns the value of a trace event unsigned long long integer argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_ulong_long` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

unsigned long long integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_ulong_long()" on page 16-41
- "start_blk_arg_ulong_long()" on page 16-83
- "offset_blk_arg_ulong_long()" on page 16-161

end_blk_arg_ushort()

DESCRIPTION

The `end_blk_arg_ushort()` function returns the value of a trace event unsigned short integer argument located at a particular byte offset in the argument space associated with the event associated with the *end event* of the *most recent instance of a state*.

SYNTAX

`end_blk_arg_ushort` (*byte_offset*[, *PR*])

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_ushort()” on page 16-42
- “start_blk_arg_ushort()” on page 16-84
- “offset_blk_arg_ushort()” on page 16-162

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*. For events recorded with `trace_event_blk()`, it returns the number of bytes recorded in the argument space.

SYNTAX

`end_num_args` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 16-43
- "start_num_args()" on page 16-85
- "end_arg()" on page 16-100
- "offset_num_args()" on page 16-163

end_pid()

DESCRIPTION

The `end_pid()` function returns the PID associated with the *end event* of the *last completed instance of a state*.

NOTE

All Linux threads within the same program share the same PID value. For trace events generated with the NightTrace Logging API, the value logged as the process identifier is the common PID. For kernel events, the value logged for the process identifier is the actually the thread's TID (see **`gettid(2)`**).

SYNTAX

`end_pid` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "pid()" on page 16-44
- "start_pid()" on page 16-86
- "offset_pid()" on page 16-164

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*. The thread identifier is that returned by the system call **`gettid(2)`**.

SYNTAX

`end_thread_id` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "thread_id()" on page 16-45
- "start_thread_id()" on page 16-87
- "offset_thread_id()" on page 16-165

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

integer

SEE ALSO

- “task_id()” on page 16-46
- “start_task_id()” on page 16-88
- “offset_task_id()” on page 16-166

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "tid()" on page 16-47
- "start_tid()" on page 16-89
- "offset_tid()" on page 16-167

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 NightTrace kernel trace event files. Kernel tracing is not supported on all operating systems. See "Kernel Dependencies" on page B-1 for more information.

SYNTAX

`end_cpu` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "cpu()" on page 16-48
- "start_cpu()" on page 16-90
- "offset_cpu()" on page 16-168

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "offset()" on page 16-49
- "start_offset()" on page 16-91

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- “time()” on page 16-50
- “start_time()” on page 16-92
- “state_gap()” on page 16-134
- “state_dur()” on page 16-135
- “offset_time()” on page 16-169

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "node_id()" on page 16-51
- "start_node_id()" on page 16-93
- "offset_node_id()" on page 16-170

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

string

SEE ALSO

- “pid_table_name()” on page 16-52
- “start_pid_table_name()” on page 16-94
- “offset_pid_table_name()” on page 16-171

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

string

SEE ALSO

- "tid_table_name()" on page 16-53
- "start_tid_table_name()" on page 16-95
- "offset_tid_table_name()" on page 16-172

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

string

SEE ALSO

- “node_name()” on page 16-54
- “start_node_name()” on page 16-96
- “offset_node_name()” on page 16-173

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 12-11.

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- "start_time()" on page 16-92
- "end_time()" on page 16-129
- "event_gap()" on page 16-58
- "state_dur()" on page 16-135

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

RETURN TYPE

double-precision floating point

SEE ALSO

- “state_gap()” on page 16-134

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 "Profile References" on page 16-193.

RETURN TYPE

integer

SEE ALSO

- "Start Functions" on page 16-60
- "summary_matches()" on page 16-183

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` [([*PR*])]

PARAMETERS

PR

A user-defined *profile reference*. 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 “Profile References” on page 16-193.

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_arg_long()`
- `offset_arg_long_dbl()`
- `offset_arg_long_long()`
- `offset_blk_arg()`
- `offset_blk_arg_bits()`
- `offset_blk_arg_char()`
- `offset_blk_arg_dbl()`
- `offset_blk_arg_flt()`
- `offset_blk_arg_long()`
- `offset_blk_arg_long_bits()`
- `offset_blk_arg_long_dbl()`
- `offset_blk_arg_long_long()`
- `offset_blk_arg_long_ubits()`
- `offset_blk_arg_short()`
- `offset_blk_arg_string()`
- `offset_blk_arg_ubits()`
- `offset_blk_arg_uchar()`
- `offset_blk_arg_uint()`
- `offset_blk_arg_ulong_long()`
- `offset_blk_arg_ushort()`
- `offset_num_args()`
- `offset_pid()`
- `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 16-49, “start_offset()” on page 16-91, “end_offset()” on page 16-128, “min_offset()” on page 16-181, and “max_offset()” on page 16-182.

offset_id()

DESCRIPTION

The `offset_id()` function returns the *trace event ID* of the ordinal trace event (*offset*).

SYNTAX

`offset_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

- "id()" on page 16-20
- "start_id()" on page 16-62
- "end_id()" on page 16-99

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

N

Specifies the *N*th argument logged with the trace event. Defaults to 1.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

integer

SEE ALSO

- "arg()" on page 16-21
- "start_arg()" on page 16-63
- "end_arg()" on page 16-100
- "offset_arg_dbl()" on page 16-142
- "offset_num_args()" on page 16-163

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

N

Specifies the *N*th argument logged with the trace event. Defaults to 1.

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

- "arg_dbl()" on page 16-22
- "start_arg_dbl()" on page 16-64
- "end_arg_dbl()" on page 16-101
- "offset_arg()" on page 16-141
- "offset_num_args()" on page 16-163

offset_arg_long()

DESCRIPTION

The `offset_arg_long()` function returns the value of a particular *trace event argument* for the ordinal trace event (*offset*).

SYNTAX

`offset_arg`[*N*]`_long` (*offset_expr*)

PARAMETERS

N

Specifies the *N*th argument logged with the trace event. Defaults to 1.

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

- "arg_long()" on page 16-23
- "start_arg_long()" on page 16-65
- "end_arg_long()" on page 16-102
- "offset_arg()" on page 16-141
- "offset_num_args()" on page 16-163

offset_arg_long_dbl()

DESCRIPTION

The `offset_arg_long_dbl()` function returns the value of a particular *trace event argument* for the ordinal trace event (*offset*).

SYNTAX

`offset_arg`[*N*]`_long_dbl` (*offset_expr*)

PARAMETERS

N

Specifies the *N*th argument logged with the *trace event*. Defaults to 1.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

long double-precision floating point

SEE ALSO

- "num_args()" on page 16-43
- "arg_long_dbl()" on page 16-24
- "start_arg_long_dbl()" on page 16-66
- "end_arg_long_dbl()" on page 16-103

offset_arg_long_long()

DESCRIPTION

The `offset_arg_long_long()` function returns the value of a particular *trace event argument* for the ordinal trace event (*offset*).

SYNTAX

`offset_arg`[*N*]`_long_long` (*offset_expr*)

PARAMETERS

N

Specifies the *N*th argument logged with the *trace event*. Defaults to 1.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

long long integer

SEE ALSO

- "num_args()" on page 16-43
- "arg_long_long()" on page 16-25
- "start_arg_long_long()" on page 16-67
- "end_arg_long_long()" on page 16-104

offset_blk_arg()

DESCRIPTION

The `offset_blk_arg()` function returns the value of a trace event argument located at a particular byte offset in the argument space associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg` (*byte_offset* , *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

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 16-43
- "blk_arg()" on page 16-26
- "start_blk_arg()" on page 16-68
- "end_blk_arg()" on page 16-105

offset_blk_arg_bits()

DESCRIPTION

The `offset_blk_arg_bits()` function returns the value of a trace event signed bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the event associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_bits` (*byte_offset*, *bit_offset*, *bit_size*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

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 16-43
- “blk_arg_bits()” on page 16-27
- “start_blk_arg_bits()” on page 16-69
- “end_blk_arg_bits()” on page 16-106

offset_blk_arg_char()

DESCRIPTION

The `offset_blk_arg_char()` function returns the value of a trace event signed character argument located at a particular byte offset in the argument space associated with the event associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_char` (*byte_offset*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

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 16-43
- "blk_arg_char()" on page 16-28
- "start_blk_arg_char()" on page 16-70
- "end_blk_arg_char()" on page 16-107

offset_blk_arg_dbl()

DESCRIPTION

The `offset_blk_arg_dbl()` function returns the value of a trace event double-precision floating point argument located at a particular byte offset in the argument space associated with the event associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_dbl` (*byte_offset*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

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

- “num_args()” on page 16-43
- “blk_arg_dbl()” on page 16-29
- “start_blk_arg_dbl()” on page 16-71
- “end_blk_arg_dbl()” on page 16-108

offset_blk_arg_flt()

DESCRIPTION

The `offset_blk_arg_flt()` function returns the value of a trace event single-precision floating point argument located at a particular byte offset in the argument space associated with the event associated with the ordinal trace event (*offset*).

SYNTAX

`offset blk_arg_flt` (*byte_offset*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

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

- "num_args()" on page 16-43
- "blk_arg_flt()" on page 16-30
- "start_blk_arg_flt()" on page 16-72
- "end_blk_arg_flt()" on page 16-109

offset_blk_arg_long()

DESCRIPTION

The `offset_blk_arg_long()` function returns the value of a trace event long integer argument located at a particular byte offset in the argument space associated with the event associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_long` (*byte_offset*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

long integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_long()" on page 16-31
- "start_blk_arg_long()" on page 16-73
- "end_blk_arg_long()" on page 16-110

offset_blk_arg_long_bits()

DESCRIPTION

The `offset_blk_arg_long_bits()` function returns the value of a trace event signed long bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the event associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_long_bits` (*byte_offset*, *bit_offset*, *bit_size*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

long integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_long_bits()" on page 16-32
- "start_blk_arg_long_bits()" on page 16-74
- "end_blk_arg_long_bits()" on page 16-111

offset_blk_arg_long_dbl()

DESCRIPTION

The `offset_blk_arg_long_dbl()` function returns the value of a trace event long double-precision floating point argument located at a particular byte offset in the argument space associated with the event associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_long_dbl` (*byte_offset*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

long double-precision floating point

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_long_dbl()" on page 16-33
- "start_blk_arg_long_dbl()" on page 16-75
- "end_blk_arg_long_dbl()" on page 16-112

offset_blk_arg_long_long()

DESCRIPTION

The `offset_blk_arg_long_long()` function returns the value of a trace event long long integer argument located at a particular byte offset in the argument space associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_long_long` (*byte_offset*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

long long integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_long_long()" on page 16-34
- "start_blk_arg_long_long()" on page 16-76
- "end_blk_arg_long_long()" on page 16-113

offset_blk_arg_long_ubits()

DESCRIPTION

The `offset_blk_arg_long_ubits()` function returns the value of a trace event unsigned long integer bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_long_ubits` (*byte_offset*, *bit_offset*, *bit_size*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

long long integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_long_ubits()" on page 16-35
- "start_blk_arg_long_ubits()" on page 16-77
- "end_blk_arg_long_ubits()" on page 16-114

offset_blk_arg_short()

DESCRIPTION

The `offset_blk_arg_short()` function returns the value of a trace event short integer argument located at a particular byte offset in the argument space associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_short` (*byte_offset*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

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 16-43
- "blk_arg_short()" on page 16-36
- "start_blk_arg_short()" on page 16-78
- "end_blk_arg_short()" on page 16-115

offset_blk_arg_string()

DESCRIPTION

The `offset_blk_arg_string()` function returns the value of a trace event null terminated string argument located at a particular byte offset in the argument space associated with the event associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_string` (*byte_offset* , *max_size* , *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk* or *trace_event_string*.

max_size

Specifies the maximum length of string that might be returned. If the arguments were recorded with *trace_event_blk*, this is also the total number of bytes allocated in the block for the string, regardless of its actual lenght.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

string

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_string()” on page 16-37
- “start_blk_arg_string()” on page 16-79
- “end_blk_arg_string()” on page 16-116

offset_blk_arg_ubits()

DESCRIPTION

The `offset_blk_arg_ubits()` function returns the value of a trace event unsigned bit field argument located at a particular byte and bit offset with a particular bit size in the argument space associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_ubits` (*byte_offset*, *bit_offset*, *bit_size*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

bit offset

Specifies the bit offset of the argument recorded with the *trace_event_blk*.

bit size

Specifies the size in bits of the argument record with *trace_event_blk*.

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 16-43
- "blk_arg_ubits()" on page 16-38
- "start_blk_arg_ubits()" on page 16-80
- "end_blk_arg_ubits()" on page 16-117

offset_blk_arg_uchar()

DESCRIPTION

The `offset_blk_arg_uchar()` function returns the value of a trace event unsigned character argument located at a particular byte offset in the argument space associated with the event associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_uchar` (*byte_offset*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

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 16-43
- “blk_arg_uchar()” on page 16-39
- “start_blk_arg_uchar()” on page 16-81
- “end_blk_arg_uchar()” on page 16-118

offset_blk_arg_uint()

DESCRIPTION

The `offset_blk_arg_uint()` function converts the unsigned integer trace event argument at a particular byte offset in the argument space associated with the ordinal trace event (*offset*) to a long.

NOTE

You can convert the long return value to an unsigned value using the cast operator. For example:

```
(unsigned long) offset_blk_arg_uint(0) > 0x80000000
```

SYNTAX

`offset_blk_arg_uint` (*byte_offset*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

unsigned integer

SEE ALSO

- "num_args()" on page 16-43
- "blk_arg_uint()" on page 16-40
- "start_blk_arg_uint()" on page 16-82
- "end_blk_arg_uint()" on page 16-119

offset_blk_arg_ulong_long()

DESCRIPTION

The `offset_blk_arg_ulong_long()` function returns the value of a trace event unsigned long long integer argument located at a particular byte offset in the argument space associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_ulong_long` (*byte_offset* , *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

unsigned long long integer

SEE ALSO

- “num_args()” on page 16-43
- “blk_arg_ulong_long()” on page 16-41
- “start_blk_arg_ulong_long()” on page 16-83
- “end_blk_arg_ulong_long()” on page 16-120

offset_blk_arg_ushort()

DESCRIPTION

The `offset_blk_arg_ushort()` function returns the value of a trace event unsigned short integer argument located at a particular byte offset in the argument space associated with the ordinal trace event (*offset*).

SYNTAX

`offset_blk_arg_ushort` (*byte_offset*, *offset_expr*)

PARAMETERS

byte offset

Specifies the byte offset of the argument recorded with *trace_event_blk*.

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 16-43
- "blk_arg_ushort()" on page 16-42
- "start_blk_arg_ushort()" on page 16-84
- "end_blk_arg_ushort()" on page 16-121

offset_num_args()

DESCRIPTION

The `offset_num_args()` function returns the number of arguments logged with the ordinal trace event (*offset*). For events recorded with `trace_event_blk()`, it returns the number of bytes recorded in the argument space.

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 16-43
- “start_num_args()” on page 16-85
- “end_num_args()” on page 16-122
- “offset_arg()” on page 16-141
- “offset_arg_dbl()” on page 16-142

offset_pid()

DESCRIPTION

The `offset_pid()` function returns the PID from which the ordinal trace event (*offset*) was logged.

NOTE

All Linux threads within the same program share the same PID value. For trace events generated with the NightTrace Logging API, the value logged as the process identifier is the common PID. For kernel events, the value logged for the process identifier is the actually the thread's TID (see **`gettid(2)`**).

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 16-44
- "start_pid()" on page 16-86
- "end_pid()" on page 16-123

offset_thread_id()

DESCRIPTION

The `offset_thread_id()` function returns the *thread* identifier from which the ordinal trace event (*offset*) was logged. The thread identifier is the value returned from the system call **`gettid(2)`**.

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 16-45
- “start_thread_id()” on page 16-87
- “end_thread_id()” on page 16-124

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 16-46
- "start_task_id()" on page 16-88
- "end_task_id()" on page 16-125

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

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

integer

SEE ALSO

- “tid()” on page 16-47
- “start_tid()” on page 16-89
- “end_tid()” on page 16-126

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 NightTrace kernel trace event files. Kernel tracing is not supported on all operating systems. See "Kernel Dependencies" on page B-1 for more information.

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 16-48
- "start_cpu()" on page 16-90
- "end_cpu()" on page 16-127

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 16-50
- “start_time()” on page 16-92
- “end_time()” on page 16-129

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

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

integer

SEE ALSO

- "node_id()" on page 16-51
- "start_node_id()" on page 16-93
- "end_node_id()" on page 16-130

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 16-52
- “start_pid_table_name()” on page 16-94
- “end_pid_table_name()” on page 16-131

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 16-53
- "start_tid_table_name()" on page 16-95
- "end_tid_table_name()" on page 16-132

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 16-54
- “start_node_name()” on page 16-96
- “end_node_name()” on page 16-133

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

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

string

SEE ALSO

- "process_name()" on page 16-55

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 16-56

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

offset_expr

An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

string

SEE ALSO

- "thread_name()" on page 16-57

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 16-177

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 16-177

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 16-177

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 16-177

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 16-177

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 16-177

summary_matches()

DESCRIPTION

The `summary_matches()` function returns the number of times the summary criteria was matched in the time range.

SYNTAX

```
summary_matches ()
```

RETURN TYPE

integer

SEE ALSO

- “event_matches()” on page 16-59
- “state_matches()” on page 16-136

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()`
- `lookup_pc()`

For more information about tables, see "Tables" on page 7-14 and "Kernel String Tables" on page 17-17.

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`, and `device`. For more information on these tables, see "Pre-Defined Strings Tables" on page 7-17 and "Kernel String Tables" on page 17-17.

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 Condition, Start Condition, or End Condition of a display object configuration
- A Condition, Start Condition, or End Condition of a Profile configuration
- An Output Text field of a Data Box
- 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 7-15.

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`, and `device`. For more information on these tables, see "Kernel String Tables" on page 17-17.

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 profiles, 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 7-15):

```
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 Condition when configuring a Data Box (see "Data Graph" on page 12-12):

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 7-15.

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 Output Text field of a Data Box configuration 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 7-20.

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(3)`** 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 Output Text field of a Data Box. 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.

lookup_pc()

The `lookup_pc()` routine returns the location of a program counter in the specified executable file.

SYNTAX

`char * lookup_pc (long` *pc_value*`, char *` *executable_file_path*`)`

PARAMETERS

pc_value

the address pointer value of the instruction to be located.

executable_file_path

the path of the executable file containing the pc.

DESCRIPTION

This function can be used in expressions, typically in `format()` statements.

Given a PC value, it returns a string describing the location of the PC in the specified executable file. The string returned includes the name of the routine containing it and the file and line number associated with the PC, depending on how much symbolic and debug information is available in the file.

NightTrace attempts to locate the executable using the specified *executable_file_path*. If the specified path is a simple file name without a directory indication, NightTrace will first attempt to match the file's specified simple name with those of any executables given on the command line. Otherwise, NightTrace will attempt to locate the file exactly as specified. For example,

```
ntrace /tmp/a.out
...
format ("My PC is %s", lookup_pc(arg1,"a.out"))
```

will refer to `/tmp/a.out`, whereas

```
format ("My PC is %s", lookup_pc(arg1,"./a.out"))
```

will reference `$PWD/./a.out`.

A handy way to use `lookup_pc` is to use the built-in NightTrace function `process_name()`. For example:

```
format ("My PC is %s", lookup_pc(arg1,process_name()))
```

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

RETURN TYPES

A string is always returned from `lookup_pc()` regardless of whether it can locate the specified file or can obtain symbolic information from it. At a minimum, the string returned includes the address passed in as *pc_value* in hexadecimal notation.

Profile References

Profile references provide a means for referencing a set of one or more trace events which may be restricted by conditions specified by the user.

Profile references can be used within trace event functions (see “Trace Event Functions” on page 16-18).

A profile reference is simply the name of the profile.

Profiles are created and managed using the Profiles Definition panel (see “Profile Definition Panel” on page 13-1 for more information).

17
Kernel Tracing

This chapter provides an introduction to kernel tracing. It also discusses the steps required to produce a highly detailed picture of kernel activity with NightTrace. You can customize the default NightTrace kernel timelines or combine kernel information with user-application trace information.

NOTE

Not all operating system distributions support NightTrace kernel tracing. See “Kernel Dependencies” on page B-1 for more information.

NightTrace transforms the raw kernel events as defined in **/usr/include/linux/tracer.h** to NightTrace events. The raw kernel event numbers are biased by the value 4300 to form the NightTrace event ID number. Normally, the arguments logged with the raw kernel events are directly converted to integer-sized NightTrace arguments. There are some exceptions which are noted in this chapter.

Primary Kernel Trace Events

The following kernel trace events are of primary interest:

- SCHEDCHANGE
- SYSCALL_ENTRY,SYSCALL_EXIT, SYSCALL_SUSPEND, and SYSCALL_RESUME
- IRQ_ENTRY, IRQ_EXIT, SOFT_IRQ_ENTRY, and SOFT_IRQ_EXIT
- TRAP_ENTRY, TRAP_EXIT, TRAP_SUSPEND, and TRAP_RESUME
- PROCESS, NETWORK, and MEMORY

These trace events and several others are enabled by default when starting a kernel trace daemon. You can change the default enabled event set in **ntrace** in the Enabled Events area of the Edit Daemon Definition dialog or using **-events** command line option to **ntracekd**.

The following sections discuss the primary trace events.

Context Switch Trace Event

There is only one context switch trace event:

`SCHEDCHANGE` *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 process identifier (PID) of the process being switched in. This information is somewhat 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.

This identifier is identical to the return value of the **`gettid(2)`** system call. See "pid()" on page 16-44.

NOTE:

The `SCHEDCHANGE` event argument differs from the argument logged with the corresponding raw kernel event as described in **`/usr/include/linux/tracer.h`.**

Interrupt Trace Events

There are two trace events associated with machine interrupts:

`IRQ_ENTRY` *arg1 arg2 arg3*

This trace event is logged whenever an interrupt occurs. It has three arguments:

arg1

Reserved for future use

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 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 17-17.

`IRQ_EXIT` *arg1 arg2 arg3*

This trace event is logged whenever an interrupt is exited. Its arguments are identical to those of the `IRQ_ENTRY` trace event.

NOTE:

The `IRQ_ENTRY` and `IRQ_EXIT` event arguments differ from their raw kernel counterparts as described in **`/usr/include/linux/tracer.h`**.

Additional exception processing is done on behalf of the kernel by kernel daemons that run as user-level processes. Such exception processing is identified by the following two events:

`SOFT_IRQ_ENTRY` *arg1 arg2*
`SOFT_IRQ_EXIT`

These event pairs surround soft interrupt processing and are usually associated with a **`ksoftirq`** daemon process.

The arguments logged with `SOFT_IRQ_ENTRY` are internal kernel parameters which are explained in **`/usr/include/linux/tracer.h`**.

Exception Trace Events

There are four trace events associated with exceptions:

`TRAP_ENTRY` *arg1 arg2 arg3*

This trace event is logged whenever a machine exception occurs. It has three arguments:

arg1

This argument contains the value of 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 17-17.

arg2

This argument contains the value of the program counter where the exception occurred.

arg3

This argument contains the value of the faulting address, for those exception types which involved virtual memory faults.

TRAP_EXIT *arg1*

This trace event is logged whenever exception processing is completed. It has one argument that is identical to the first argument that is logged with the TRAP_ENTRY trace event.

TRAP_SUSPEND *arg1*
TRAP_RESUME *arg1*

These trace events are logged when exception processing is suspended before it is completed, and subsequently resumed. A TRACE_SUSPEND event will be followed immediately by a SCHEDCHANGE event which signifies a context switch to another process while the process that caused the exception is blocked pending exception processing completion. The single argument logged for both events is the exception vector number associated with the originating TRAP_ENTRY event.

Syscall Trace Events

There are four trace events associated with system calls:

SYSCALL_ENTRY *arg1 arg2 arg3*

This trace event is logged whenever a system call is entered. It has three arguments:

arg1

This argument is the value of the program counter from which the system call was made. Depending on the system type, this value may not be particularly useful as many system calls occur from the same page in virtual memory, commonly referred to as the *fast system call* page.

arg2

This argument is the value of the system call number that identifies the system call. This is an index into the pre-defined syscall string table.

arg3

This argument is the value of the device number that indicates the type of device that is associated with the system call, 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 17-17.

SYSCALL_EXIT *arg1 arg2 arg3*

This trace event is logged whenever a system call is completed. It has three arguments; the second and third arguments are identical to the second and third arguments logged with the originating SYSCALL_ENTRY trace event. The first argument is the value returned by the system call.

NOTE:

The return value of the system call is only available on RedHawk version 2.3 and beyond. On previous versions, the value will be zero, regardless of the success or failure of the system call.

`SYSCALL_SUSPEND` *arg1 arg2 arg3*
`SYSCALL_RESUME` *arg1 arg2 arg3*

These trace events are logged when system call processing is suspended before it is completed, and subsequently resumed. A `SYSCALL_SUSPEND` event will be followed immediately by a `SCHEDCHANGE` event which signifies a context switch to another process while the process that executed the system call is blocked pending system call processing completion. The arguments logged for both events are identical to the arguments associated with the originating `SYSCALL_ENTRY` event.

NOTE:

The `SYSCALL_ENTRY` and `SYSCALL_EXIT` event arguments differ from their raw kernel counterparts as described in **`/usr/include/linux/tracer.h`**.

Kernel Work Events

Kernel work events occur during system calls, exceptions, and interrupt processing. They include the following events:

`PROCESS` *arg1 arg2 arg3*

The `PROCESS` event represents process creation, exit, and signalling events. The following arguments provide detail:

arg1

This argument is an event code specific to PROCESS events as defined by **/usr/include/linux/tracer.h**. The codes and their meanings are described in the Table 17-1:

Table 17-1. PROCESS Event Codes

Code	Meaning
1	Kernel thread creation
2	Process creation (fork or clone)
3	Process exit
4	Process wait
5	Process signal
6	Process wake-up

arg2

The meaning of this argument is dependent on the value of *arg1*. Normally, this argument is the process ID of the process associated with the event. However, when a signal is sent, this argument is the signal number.

arg3

The meaning of this argument is dependent on the value of *arg1*. Normally, this argument is the value of an internal kernel function pointer. However, when a signal is sent, this argument is the process ID of the process being signalled.

NETWORK

This event is logged to indicate networking activity.

arg1

This argument is an event code specific to NETWORK events as defined by **/usr/include/linux/tracer.h**. The codes and their meanings are described in Table 17-2:

Table 17-2. NETWORK Kernel Event Sub-ID Codes

Code	Meaning
1	A packet was received
2	A packet was sent

arg2

This argument is an internal kernel data value associated with the event.

`MEMORY`

This event is logged to indicate a variety of virtual memory events.

arg1

This argument is an event code specific to `MEMORY` events as defined by **`/usr/include/linux/tracer.h`**. The codes and their meanings are described in Table 17-3:

Table 17-3. MEMORY Kernel Event Sub-ID Codes

Code	Meaning
1	Allocating pages
2	Freeing pages
3	Swapping in pages
4	Swapping out pages
5	Start to wait for page
6	End waiting for page

arg2

This argument is an internal kernel data value associated with the event.

Additional Kernel Events

There are many more kernel events that occur other than those described in the sections above. They are defined by the enumerated type `event_id` in the **`/usr/include/linux/tracer.h`** header file. Not all events defined in that file are enabled by default.

For many kernel events, a corresponding structure is defined. The content of the structure contains additional detail describing the event. The structure is unpacked into individual arguments which are logged with the event. As many integer arguments are logged as required to cover the size of the structure.

For example, an `IPC` kernel event includes data in the following structure, as defined by **`/usr/include/linux/tracer.h`**:

```
/* TRACE_EV_IPC */
typedef struct {
    unsigned int event_sub_id;
    unsigned int event_data1;
    unsigned int event_data2;
}  trace_ipc;
```

The following arguments are logged with an `IPC` event:

arg1

This first word of the structure -- `event_sub_id`

arg2

The second word of the structure -- `event_data1`

arg2

The third word of the structure -- `event_data2`

The kernel includes a `CUSTOM` event which can contain dynamically-sized data. This flexible unpacking scheme allows new dynamically-sized events to be created and logged effectively by NightTrace.

Logging Custom Kernel Events

The `CUSTOM` event is not enabled by default in kernel trace daemons. You can change the default enabled event set in **`ntrace`** in the Events area of the Edit Daemon Definition dialog or using the **`--events`** command line option to **`ntracekd`**, e.g:

```
ntracekd --size=20M --events=+CUSTOM data-file
```

From User Programs

User programs can log `CUSTOM` kernel trace events with `ioctl` calls.

The following structure is defined in **`/usr/include/ntrace.h`**:

```
typedef struct {
    unsigned int id;        // Custom event ID
    unsigned int data_size; // Size of optional data
    void * data;            // Optional data
} nt_trace_custom;
```

The following code fragment provides an example of how to log a custom kernel event from a user application:

```
#include <ntrace.h>
#include <fcntl.h>
...
{ int fd;
  int err;
  typedef struct {
      int i;
      int j;
      double d;
  } my_data_t;
  my_data_t data = { 47, 0, 3.14159 };
  nt_trace_custom event;
  event.id = 17;
  event.data_size = sizeof(data);
  event.data = &data;
  fd = open ("/dev/tracer", O_RDWR, 0);
  err = ioctl(fd,NT_TRACER_LOG_CUSTOM_EVENT,&event)==-1;
  close(fd);
};
```

From Kernel Modules

The following code fragment provides an example of how to insert `CUSTOM` kernel trace events inside kernel code; for example, a kernel module.

```
#include <linux/tracer.h>
…
typedef struct {
    int i;
    int j;
    double d;
} my_data_t;
my_data_t data = { 47, 0, 3.14159 };
TRACE_CUSTOM(17,&data,sizeof(data))
```

Retrieving Custom Events

Custom events are always logged with the trace event ID of "CUSTOM", which is the value 4319.

A minimum of three data values are always logged with it; these correspond to the components of the following structure defined in **`/usr/include/linux/tracer.h`**:

```
typedef struct {
   unsigned int id;         // Custom Event ID
   unsigned int data_size;  // Size of data recorded by event
   void *       data;       // Data recorded by event
} trace_custom;
```

This structure corresponds directly to `nt_trace_custom` from the example under "From User Programs" on page 17-9 and the arguments to the `TRACE_CUSTOM` call in the example under "From Kernel Modules" on page 17-9 (although the order of the arguments in the Kernel Modules example differs from the order of the components).

The additional data logged with the event immediately follows as additional values. The entire set of values, those from the `trace_custom` structure and those from additional supplied data items if any, are logged as a continual block of memory.

Note that the actual value of the `trace_custom.data` component is not very interesting from within **`ntrace`**. The actually data it originally pointed to now immediately follows the `trace_custom.data` component in memory.

Extracting the values of interest within **`ntrace`** is best done with the `blk_arg` family of NightTrace functions. These functions all take a byte offset as their first argument. The function name itself defines the type (and therefore the size) of the data value to be extracted; for example, `blk_arg_dbl` extracts double-precision floating point.

The following table shows NightTrace expressions and their corresponding value for the event logged in both examples above (the examples were constructed so that they effectively logged the same data values):

Expressions		Value	Comment
i386	**x86_64**		
`blk_arg(0)`	`blk_arg(0)`	17	This corresponds to the `event.id` component in the User Program example and the first argument to `TRACE_CUSTOM` in the Kernel Modules example.
`blk_arg(12)`	`blk_arg(16)`	47	This corresponds to the `data.i` component in both examples.
`blk_arg_dbl(20)`	`blk_arg_dbl(24)`	3.14159	This corresponds to the `data.d` component in both examples.

The offsets supplied in the `blk_arg*` expressions differ between architectures; the size of the `void*` component of `trace_custom` is 4 bytes on i386 systems but 8 bytes on x86_64 systems.

NOTE

By default within **`ntrace`**, the entire block of memory is displayed as a series of integer-sized arguments since the layout of the additional data items is unknown to NightTrace.

Viewing Kernel Trace Event Files

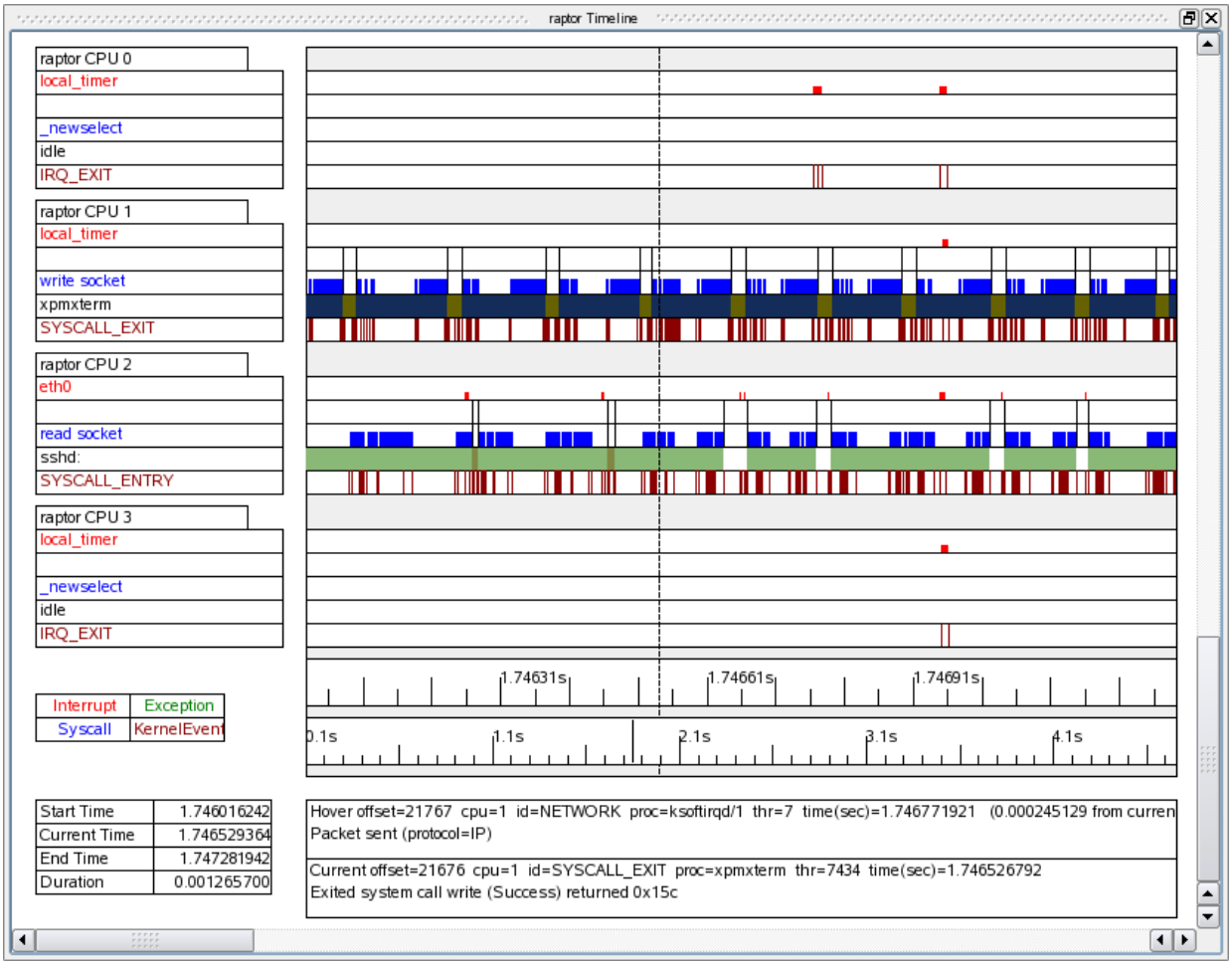
NightTrace automatically builds kernel timelines when **`ntrace`** is invoked with kernel data (see “Kernel Timelines” on page 17-12). The number of CPUs is detected from the kernel trace data and controls how the page is built.

In addition, you may customize a kernel timeline using the Build Custom Kernel Timeline dialog (see “Custom Kernel Timeline...” on page 8-20) which is accessed by selecting the Custom Kernel Timeline… menu item from the Timelines menu on the NightTrace Main Window (see “Custom Kernel Timeline...” on page 8-20).

Kernel Timelines

Figure 17-1 shows a sample kernel timeline for a quad CPU system.

Figure 17-1. Sample Kernel timeline



For each CPU, several rows of information are displayed. The position of the current time line determines the values that appear on the kernel timelines. 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 or at the current time line.

The following sections discuss all of the different pieces of information in detail

- "Node and CPU Information" on page 17-13
- "Context Switch Information" on page 17-13
- "Interrupt Information" on page 17-14
- "Exception Information" on page 17-14
- "System Call Information" on page 17-15

- "Process Information" on page 17-16
- "Kernel Events" on page 17-16
- "Color Information" on page 17-17

Node and CPU Information

Figure 17-2 shows the Grid Label (see "Label" on page 12-20) that appears on kernel timelines which displays information about the node and CPU corresponding to the trace data being displayed.

Figure 17-2. 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.

The **`cpu(1)`** command displays the relationship of physical CPU numbers to logical CPU numbers, but since most all interfaces use logical CPU numbers, it is not normally of significant interest.

Context Switch Information

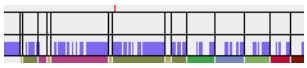


Figure 17-3. Context Switch Lines

Figure 17-3 shows an example of several context switch lines. *Context switch lines* are superimposed on the exception and system call 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 Process Information box: the Process Information box shows the process associated with the context switch line that immediately precedes the current time line.

Interrupt Information

Figure 17-4. Interrupt Box and Interrupt Graph

Figure 17-4 shows an 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 interrupt box displays the name of the last interrupt prior to or immediately at 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 interrupt box will update to display the name of the interrupt.

Because the 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 interrupt box contains a valid interrupt name. This 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 timer interrupt, usually once every 10 milliseconds. The interrupt box is a Data Box ("Data Box" on page 12-20) and the interrupt graph is a Data Graph ("Data Graph" on page 12-12). See "Creating Timeline Objects" on page 12-8 for more information on configuring Data Boxes and Data Graphs.

Exception Information

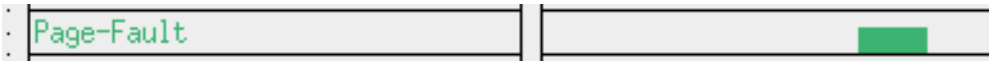


Figure 17-5. Exception Box and Exception Graph

Figure 17-5 shows a 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 17-5.

The exception box displays the last exception prior to or at 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 exception box will update to display the name of the exception.

Because the 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 exception box contains a valid exception name. This signifies that the last exception on the CPU ended prior to the beginning of the current interval.

The exception box is a Data Box (“Data Box” on page 12-20) and the last exception graph is a State Graph (see “State Graph” on page 12-11). See “Creating Timeline Objects” on page 12-8 for more information on creating and configuring Data Boxes and State Graphs.

System Call Information

Figure 17-6. System Call Box and System Call Graph

Figure 17-6 shows a system call box and a system call graph. The system call graph displays a state that is drawn whenever a system call is executing on the associated CPU. Unlike interrupts, system calls cannot nest, so they are always graphed with the same height.

Context switch lines are superimposed on system call graphs. It is common to see a context switch line at what looks like the very end (or beginning) of a system call. Usually, this does not indicate that the system call has ended, only that it has been suspended because the process that originated the system call has switched out. The system call resumes when the process is switched back in again. An example of a system call being suspended and resumed can be seen at the right end of the system call graph in the figure.

The system call box displays the last system call prior to or at the current time line that executed (and may still be executing) on the associated CPU. If the system call is associated with a device, the name of the device is shown after the name of the system call.

The system call box can be used with the system call graph to identify any system calls that are currently visible on the graph. Simply move the current time line onto a graphed system call, and the system call box will update to display the name of the system call.

Because the system call box displays the name of the last system call that executed, it is possible for there to be no system calls visible on the system call graph even though the system call box contains a valid system call name. This signifies that the last system call on the CPU ended prior to the beginning of the current interval.

It is possible for the first system call logged by a process since kernel tracing began to be unknown. This can occur if the process is switched in and immediately resumes a system call that was previously suspended. If this occurs, the system call box will display “`can't determine`” for the name of the system call.

The system call box is a Data Box (see "Data Box" on page 12-20), and the last system call graph is a State Graph (see "State Graph" on page 12-11). See "Creating Timeline Objects" on page 12-8 for more information on configuring Data Boxes and State Graphs.

Process Information

Figure 17-7. Process Information Row

Figure 17-7 shows the Process Information row which includes a process data box (see "Data Graph" on page 12-12) and a process state graph (see "State Graph" on page 12-11). See "Creating Timeline Objects" on page 12-8 for more information on creating and configuring Data Boxes and State Graphs.

The data box indicates the name of the process (other than **`/idle`**) that last executed on the CPU prior to or at the current timeline.

The state graph uses multi-colored states to indicate when a process other than **`/idle`** is executing on a CPU. The colors are assigned by NightTrace using a heuristic that takes into account all processes represented by the data set. You cannot predict which color will be associated with a specific process, but once the color is assigned, it remains constant throughout the current NightTrace session.

Kernel Events

Figure 17-8. Kernel Events Row

Figure 17-8 shows the Kernel Events row which includes a kernel event data box (see "Data Box" on page 12-20) and a kernel event graph (see "Event Graph" on page 12-10). See "Creating Timeline Objects" on page 12-8 for more information on creating and configuring Data Boxes and Event Graphs.

The data box indicates the name of the last kernel event logged for that CPU prior to or at the current timeline.

The event graph shows a vertical line for every kernel event.

Color Information

Figure 17-9. Color Key

Figure 17-9 shows the color key that is located on the bottom left of the grid on the pre-defined kernel timelines.

The text in the color key is color-coded. By default, the word "Interrupt" is red, and all display objects on the kernel timeline 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 system calls are also blue. By default, the word "KernelEvent" is dark red, and all display objects that display kernel events in that row are dark red.

Currently, the default colors cannot be modified. Setting color preferences will be provided in a future update.

Kernel String Tables

There are nine 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, "ide0")
```

`syscall`

This string table contains the names of all the possible system calls that can occur on the system. 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, 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")
```

Note that using the NightTrace function `process_name()` is more convenient than having to dynamically locate and index the correct `pid_nodename` table to get the current process name.

For example, the following two expressions are equivalent:

```
process_name()
get_string(get_string(name_pid,node_id()),pid())
```

`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 timelines. For more information on the `pid` string table, see "Pre-Defined Strings Tables" on page 7-17.

18
Using the NightTrace Analysis API

The NightTrace graphical user interface is one of the primary tools for analyzing trace data (see "The NightTrace Main Window" on page 8-1). 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 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 "NightTrace Analysis API Examples" on page E-1).

NightTrace Analysis Application Programming Interface

The NightTrace Analysis Application Programming Interface consists of a number of data structures (see "Data Structures" on page 18-2) and functions (see "Functions" on page 18-9).

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_arg_t` (see "tr_arg_t" on page 18-2)
- `tr_cb_t` (see "tr_cb_t" on page 18-3)
- `tr_cond_cb_func_t` (see "tr_cond_cb_func_t" on page 18-3)
- `tr_cond_func_t` (see "tr_cond_func_t" on page 18-4)
- `tr_cond_t` (see "tr_cond_t" on page 18-4)
- `tr_dir_t` (see "tr_dir_t" on page 18-4)
- `tr_offset_t` (see "tr_offset_t" on page 18-4)
- `tr_state_action_t` (see "tr_state_action_t" on page 18-5)
- `tr_state_cb_func_t` (see "tr_state_cb_func_t" on page 18-5)
- `tr_state_info_t` (see "tr_state_info_t" on page 18-6)
- `tr_state_t` (see "tr_state_t" on page 18-7)
- `tr_stream_event_t` (see "tr_stream_event_t" on page 18-7)
- `tr_stream_func_t` (see "tr_stream_func_t" on page 18-7)
- `tr_string_node_t` (see "tr_string_node_t" on page 18-7)
- `tr_t` (see "tr_t" on page 18-8)

See "Functions" on page 18-9 for information about the functions available in the NightTrace Analysis API.

tr_arg_t

`tr_arg_t` is defined as:

```
typedef enum { int_arg,
               long_arg,
               dbl_arg,
               long_dbl_arg,
               string_arg,
               long_long_arg } tr_arg_t;
```

See "Data Structures" on page 18-2 for other data structures included in the NightTrace Analysis API.

tr_cb_t

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

```
typedef int tr_cb_t;
```

See “Data Structures” on page 18-2 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

t

data set handle

c

handle of the condition associated with this call

offset

offset of the trace event satisfying the condition

occurrence

number of times the condition has been satisfied thus far

context

user-defined field specified when the callback is defined

disable

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 18-2 for other data structures included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_offset_t" on page 18-4

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 18-2 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 18-2 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 18-2 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 18-2 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 18-2 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

t

data set handle

state

handle of the state associated with this call

offset

offset of the trace event satisfying the condition

occurrence

number of times the condition has been satisfied thus far

context

user-defined field specified when the callback is defined

disable

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 18-2 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:

`start_offset`

offset of the event that started the specified state

`end_offset`

offset of the event that ended the specified state

`gap`

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)

`duration`

time in seconds during which the specified state was active

`count`

number of completed instances of the specified state

See “Data Structures” on page 18-2 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 18-2 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;
```

NOTE

The `tr_stream_overflow` event has been deprecated and no longer occurs.

See “Data Structures” on page 18-2 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 18-2 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 18-2 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 18-2 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” on page 18-14
- “Error Detection, Collection, and Reporting” on page 18-16
- “Input Specification and Streaming Control” on page 18-18
- “Event Offset Positioning” on page 18-25
- “Basic Event Attribute Functions” on page 18-30
- “Conditions” on page 18-90
- “State-oriented Interfaces” on page 18-122
- “Output Function” on page 18-138
- “String Table Functions” on page 18-140
- “Callback Interfaces” on page 18-145

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

- `tr_activate()` (see “tr_activate()” on page 18-133)
- `tr_append_table()` (see “tr_append_table()” on page 18-143)
- `tr_arg_dbl()` (see “tr_arg_dbl()” on page 18-38)
- `tr_arg_dbl_()` (see “tr_arg_dbl_()” on page 18-45)
- `tr_arg_int()` (see “tr_arg_int()” on page 18-36)
- `tr_arg_int_()` (see “tr_arg_int_()” on page 18-44)
- `tr_argtype()` (see “tr_argtype()” on page 18-50)
- `tr_argtype_()` (see “tr_argtype_()” on page 18-51)
- `tr_blk_arg()` (see “tr_blk_arg()” on page 18-51)
- `tr_blk_arg_()` (see “tr_blk_arg_()” on page 18-52)
- `tr_blk_arg_bits()` (see “tr_blk_arg_bits()” on page 18-53)
- `tr_blk_arg_bits_()` (see “tr_blk_arg_bits_()” on page 18-54)
- `tr_blk_arg_char()` (see “tr_blk_arg_char()” on page 18-55)
- `tr_blk_arg_char_()` (see “tr_blk_arg_char_()” on page 18-55)
- `tr_blk_arg_dbl()` (see “tr_blk_arg_dbl()” on page 18-56)
- `tr_blk_arg_dbl_()` (see “tr_blk_arg_dbl_()” on page 18-57)
- `tr_blk_arg_flt()` (see “tr_blk_arg_flt()” on page 18-58)

- `tr_blk_arg_flt_()` (see "tr_blk_arg_flt_()" on page 18-58)
- `tr_blk_arg_long()` (see "tr_blk_arg_long()" on page 18-59)
- `tr_blk_arg_long_()` (see "tr_blk_arg_long_()" on page 18-60)
- `tr_blk_arg_long_bits()` (see "tr_blk_arg_long_bits()" on page 18-61)
- `tr_blk_arg_long_bits_()` (see "tr_blk_arg_long_bits_()" on page 18-62)
- `tr_blk_arg_long_dbl()` (see "tr_blk_arg_long_dbl()" on page 18-63)
- `tr_blk_arg_long_dbl_()` (see "tr_blk_arg_long_dbl_()" on page 18-63)
- `tr_blk_arg_long_long()` (see "tr_blk_arg_long_long()" on page 18-64)
- `tr_blk_arg_long_long_()` (see "tr_blk_arg_long_()" on page 18-60)
- `tr_blk_arg_long_ubits()` (see "tr_blk_arg_long_ubits()" on page 18-66)
- `tr_blk_arg_long_ubits_()` (see "tr_blk_arg_long_ubits_()" on page 18-67)
- `tr_blk_arg_short()` (see "tr_blk_arg_short()" on page 18-68)
- `tr_blk_arg_short_()` (see "tr_blk_arg_short_()" on page 18-68)
- `tr_blk_arg_string()` (see "tr_blk_arg_string()" on page 18-69)
- `tr_blk_arg_string_()` (see "tr_blk_arg_string_()" on page 18-70)
- `tr_blk_arg_ubits()` (see "tr_blk_arg_ubits()" on page 18-71)
- `tr_blk_arg_ubits_()` (see "tr_blk_arg_ubits_()" on page 18-72)
- `tr_blk_arg_uchar()` (see "tr_blk_arg_uchar()" on page 18-73)
- `tr_blk_arg_uchar_()` (see "tr_blk_arg_uchar_()" on page 18-74)
- `tr_blk_arg_ushort()` (see "tr_blk_arg_ushort()" on page 18-75)
- `tr_blk_arg_ushort_()` (see "tr_blk_arg_ushort_()" on page 18-75)
- `tr_cancel_cb()` (see "tr_cancel_cb()" on page 18-146)
- `tr_close()` (see "tr_close()" on page 18-20)
- `tr_cond_and()` (see "tr_cond_and()" on page 18-115)
- `tr_cond_cb()` (see "tr_cond_cb()" on page 18-147)
- `tr_cond_copy()` (see "tr_cond_copy()" on page 18-116)
- `tr_cond_cpu()` (see "tr_cond_cpu()" on page 18-96)
- `tr_cond_cpu_clear()` (see "tr_cond_cpu_clear()" on page 18-97)

- `tr_cond_create()` (see “tr_cond_create()” on page 18-91)
- `tr_cond_expr_and()` (see “tr_cond_expr_and()” on page 18-111)
- `tr_cond_expr_or()` (see “tr_cond_expr_or()” on page 18-112)
- `tr_cond_find()` (see “tr_cond_find()” on page 18-92)
- `tr_cond_func_and()` (see “tr_cond_func_and()” on page 18-108)
- `tr_cond_func_clear()` (see “tr_cond_func_clear()” on page 18-110)
- `tr_cond_func_or()` (see “tr_cond_func_or()” on page 18-106)
- `tr_cond_id()` (see “tr_cond_id()” on page 18-93)
- `tr_cond_id_clear()` (see “tr_cond_id_clear()” on page 18-95)
- `tr_cond_id_range()` (see “tr_cond_id_range()” on page 18-94)
- `tr_cond_name()` (see “tr_cond_name()” on page 18-118)
- `tr_cond_node()` (see “tr_cond_node()” on page 18-104)
- `tr_cond_node_clear()` (see “tr_cond_node_clear()” on page 18-105)
- `tr_cond_not()` (see “tr_cond_not()” on page 18-113)
- `tr_cond_offset()` (see “tr_cond_offset()” on page 18-121)
- `tr_cond_or()` (see “tr_cond_or()” on page 18-114)
- `tr_cond_pid()` (see “tr_cond_pid()” on page 18-98)
- `tr_cond_pid_clear()` (see “tr_cond_pid_clear()” on page 18-100)
- `tr_cond_pid_name()` (see “tr_cond_pid_name()” on page 18-99)
- `tr_cond_register()` (see “tr_cond_register()” on page 18-120)
- `tr_cond_reset()` (see “tr_cond_reset()” on page 18-92)
- `tr_cond_satisfy()` (see “tr_cond_satisfy()” on page 18-118)
- `tr_cond_satisfy_()` (see “tr_cond_satisfy_()” on page 18-119)
- `tr_cond_tid()` (see “tr_cond_tid()” on page 18-101)
- `tr_cond_tid_clear()` (see “tr_cond_tid_clear()” on page 18-103)
- `tr_cond_tid_name()` (see “tr_cond_tid_name()” on page 18-102)
- `tr_copy_input()` (see “tr_copy_input()” on page 18-138)
- `tr_copy_input_range()` (see “tr_copy_input_range()” on page 18-139)
- `tr_cpu()` (see “tr_cpu()” on page 18-82)
- `tr_cpu_()` (see “tr_cpu_()” on page 18-83)
- `tr_create_table()` (see “tr_create_table()” on page 18-142)
- `tr_destroy()` (see “tr_destroy()” on page 18-14)

- `tr_error_check()` (see "tr_error_check()" on page 18-17)
- `tr_error_clear()` (see "tr_error_clear()" on page 18-16)
- `tr_free()` (see "tr_free()" on page 18-24)
- `tr_get_item()` (see "tr_get_item()" on page 18-141)
- `tr_get_string()` (see "tr_get_string()" on page 18-140)
- `tr_halt()` (see "tr_halt()" on page 18-146)
- `tr_id()` (see "tr_id()" on page 18-32)
- `tr_id_()` (see "tr_id_()" on page 18-32)
- `tr_init()` (see "tr_init()" on page 18-14)
- `tr_iterate()` (see "tr_iterate()" on page 18-145)
- `tr_nargs()` (see "tr_nargs()" on page 18-35)
- `tr_nargs_()` (see "tr_nargs_()" on page 18-35)
- `tr_next_event()` (see "tr_next_event()" on page 18-25)
- `tr_next_event_()` (see "tr_next_event_()" on page 18-26)
- `tr_node()` (see "tr_node()" on page 18-84)
- `tr_node_()` (see "tr_node_()" on page 18-84)
- `tr_open_file()` (see "tr_open_file()" on page 18-18)
- `tr_open_stream()` (see "tr_open_stream()" on page 18-19)
- `tr_pid()` (see "tr_pid()" on page 18-76)
- `tr_pid_()` (see "tr_pid_()" on page 18-77)
- `tr_prev_event()` (see "tr_prev_event()" on page 18-26)
- `tr_prev_event_()` (see "tr_prev_event_()" on page 18-27)
- `tr_process_name()` (see "tr_process_name()" on page 18-85)
- `tr_process_name_()` (see "tr_process_name_()" on page 18-86)
- `tr_search()` (see "tr_search()" on page 18-28)
- `tr_seek()` (see "tr_seek()" on page 18-29)
- `tr_state_active()` (see "tr_state_active()" on page 18-136)
- `tr_state_active_()` (see "tr_state_active_()" on page 18-137)
- `tr_state_cb()` (see "tr_state_cb()" on page 18-148)
- `tr_state_create()` (see "tr_state_create()" on page 18-122)
- `tr_state_end_cond()` (see "tr_state_end_cond()" on page 18-131)
- `tr_state_end_cond_clear()` (see "tr_state_end_cond_clear()" on page 18-132)

- `tr_state_end_id()` (see “tr_state_end_id()” on page 18-127)
- `tr_state_end_id_clear()` (see “tr_state_end_id_clear()” on page 18-129)
- `tr_state_end_id_range()` (see “tr_state_end_id_range()” on page 18-128)
- `tr_state_find()` (see “tr_state_find()” on page 18-123)
- `tr_state_info()` (see “tr_state_info()” on page 18-134)
- `tr_state_info_()` (see “tr_state_info_()” on page 18-135)
- `tr_state_name()` (see “tr_state_name()” on page 18-124)
- `tr_state_start_cond()` (see “tr_state_start_cond()” on page 18-130)
- `tr_state_start_cond_clear()` (see “tr_state_start_cond_clear()” on page 18-131)
- `tr_state_start_id()` (see “tr_state_start_id()” on page 18-125)
- `tr_state_start_id_clear()` (see “tr_state_start_id_clear()” on page 18-127)
- `tr_state_start_id_range()` (see “tr_state_start_id_range()” on page 18-126)
- `tr_stream_notify()` (see “tr_stream_notify()” on page 18-21)
- `tr_stream_read()` (see “tr_stream_read()” on page 18-22)
- `tr_stream_size()` (see “tr_stream_size()” on page 18-23)
- `tr_task_id()` (see “tr_task_id()” on page 18-81)
- `tr_task_id_()` (see “tr_task_id()” on page 18-81)
- `tr_task_name()` (see “tr_task_name()” on page 18-86)
- `tr_task_name_()` (see “tr_task_name_()” on page 18-87)
- `tr_thread_id()` (see “tr_thread_id()” on page 18-79)
- `tr_thread_id_()` (see “tr_thread_id_()” on page 18-80)
- `tr_thread_name()` (see “tr_thread_name()” on page 18-88)
- `tr_thread_name_()` (see “tr_thread_name_()” on page 18-88)
- `tr_tid()` (see “tr_tid()” on page 18-78)
- `tr_tid_()` (see “tr_tid_()” on page 18-78)
- `tr_time()` (see “tr_time()” on page 18-33)
- `tr_time_()` (see “tr_time_()” on page 18-34)

API Initialization and Destruction

The functions related to API initialization and destruction are:

- `tr_init()` (see page 18-14)
- `tr_destroy()` (see page 18-14)

See "Functions" on page 18-9 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 18-14 for related functions. See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

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 18-14 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_init()” on page 18-14

Error Detection, Collection, and Reporting

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

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

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

- `tr_error_clear()` (see page 18-16)
- `tr_error_check()` (see page 18-17)

See "Functions" on page 18-9 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 18-16 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_error_check()" on page 18-17

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

t

data set handle

list

the list of errors that have occurred (since the last call to `tr_error_clear()` or the beginning of the program). For each entry in the *list*, `value` describes the error and `item` refers to `errno` (if appropriate). (See "tr_string_node_t" on page 18-7 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 18-16 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_string_node_t" on page 18-7
- "tr_error_clear()" on page 18-16

Input Specification and Streaming Control

The functions related to input specification and streaming control are:

- `tr_open_file()` (see page 18-18)
- `tr_open_stream()` (see page 18-19)
- `tr_close()` (see page 18-20)
- `tr_stream_notify()` (see page 18-21)
- `tr_stream_read()` (see page 18-22)
- `tr_stream_size()` (see page 18-23)
- `tr_free()` (see page 18-24)

See "Functions" on page 18-9 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

t

data set handle

filename

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 18-18 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_close()” on page 18-20

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

t

data set handle

fd

file descriptor providing streaming raw data

size

specifies the memory limit (in bytes) associated with events that have been read from the stream file descriptor but have not yet been consumed. This size can be dynamically adjusted via the `tr_stream_size()` function.

flags

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 18-18 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_stream_size()" on page 18-23
- "tr_close()" on page 18-20

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 18-18 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_open_file()" on page 18-18
- "tr_open_stream()" on page 18-19

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

t

data set handle

event

can be:

`tr_stream_overflow` - This event has been deprecated and no longer occurs. See `tr_stream_read()` for control over stream I/O operations.

`tr_stream_stall` - A stall occurs when there is an insufficient number of events available to form a segment for consumption.

func

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 18-18 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_stream_event_t” on page 18-7
- “tr_stream_func_t” on page 18-7
- “tr_stream_size()” on page 18-23
- “tr_stream_read()” on page 18-22

tr_stream_read()

`tr_stream_read()` reads events from the input stream until no events are currently available or until the specified maximum is reached. A segmented input approach is utilized so that the actual number of events read may exceed the specified maximum (by the minimum segments size).

This function need not be called at all. The stream of data is read automatically as events are consumed (by `tr_next_event()`, `tr_iterate()`, or `tr_copy_input()`).

This function is provided for situations where the rate at which events are generated exceeds that at which they are currently being consumed. If the consumption rate is significantly lower than the generation rate, the daemon writing the data to the stream could otherwise stall (block on the write) and data would be lost when the daemon's buffers fill. Calling `tr_stream_read()` in such situations ensures that data is read and stored internally for use when events are subsequently consumed by `tr_next_event()`, `tr_iterate()`, or `tr_copy_input()`.

SYNTAX

```
extern int tr_stream_read (tr_t t,
                           int max_events);
```

PARAMETERS

t

data set handle

max_events

maximum number of events to be read

RETURN VALUES

Returns the number of events read.

See "Input Specification and Streaming Control" on page 18-18 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_next_event()" on page 18-25
- "tr_iterate()" on page 18-145
- "tr_copy_input()" on page 18-138

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 18-18 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_open_stream()” on page 18-19

tr_free()

`tr_free()` releases the memory associated with events whose offsets are less than or equal to the specified offset, if those events have been consumed.

This function has no effect if the events have not been consumed or if events are not being saved (e.g., `tr_open_stream()` called without the `TR_STREAM_SAVE` flag value).

SYNTAX

```
extern int tr_free (tr_t t,
                    int event_offset);
```

PARAMETERS

t

data set handle

event_offset

specifies that the memory associated with events whose offsets are less than or equal to this value will be released when this function is called

RETURN VALUES

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

See "Input Specification and Streaming Control" on page 18-18 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_open_stream()" on page 18-19

Event Offset Positioning

The functions related to event offset positioning are:

- `tr_next_event()` (see page 18-25)
- `tr_next_event_()` (see page 18-26)
- `tr_prev_event()` (see page 18-26)
- `tr_prev_event_()` (see page 18-27)
- `tr_search()` (see page 18-28)
- `tr_seek()` (see page 18-29)

See “Functions” on page 18-9 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 18-25 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

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

t

data set handle

condition

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 18-25 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_offset_t" on page 18-4

tr_prev_event()

`tr_prev_event()` advances to the previous trace event.

SYNTAX

```
extern tr_offset_t tr_prev_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 before the first event in the data set.

See “Event Offset Positioning” on page 18-25 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

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

t

data set handle

condition

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 18-25 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

- "tr_cond_t" on page 18-4
- "tr_offset_t" on page 18-4

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

t

data set handle

direction

direction in which to search

condition

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 18-25 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_dir_t" on page 18-4
- "tr_cond_t" on page 18-4
- "tr_offset_t" on page 18-4

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

t

data set handle

offset

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 18-25 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

Basic Event Attribute Functions

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

- `tr_id()` (see page 18-32)
- `tr_id_()` (see page 18-32)
- `tr_time()` (see page 18-33)
- `tr_time_()` (see page 18-34)
- `tr_nargs()` (see page 18-35)
- `tr_nargs_()` (see page 18-35)
- `tr_arg_int()` (see page 18-36)
- `tr_arg_int_()` (see page 18-37)
- `tr_arg_dbl()` (see page 18-38)
- `tr_arg_dbl_()` (see page 18-38)
- `tr_blk_arg()` (see page 18-51)
- `tr_blk_arg_()` (see page 18-52)
- `tr_blk_arg_bits()` (see page 18-53)
- `tr_blk_arg_bits_()` (see page 18-54)
- `tr_blk_arg_char()` (see page 18-55)
- `tr_blk_arg_char_()` (see page 18-55)
- `tr_blk_arg_dbl()` (see page 18-56)
- `tr_blk_arg_dbl_()` (see page 18-57)
- `tr_blk_arg_flt()` (see page 18-58)
- `tr_blk_arg_flt_()` (see page 18-58)
- `tr_blk_arg_long()` (see page 18-59)
- `tr_blk_arg_long_()` (see page 18-60)
- `tr_blk_arg_long_bits()` (see page 18-61)
- `tr_blk_arg_long_bits_()` (see page 18-62)
- `tr_blk_arg_long_dbl()` (see page 18-63)
- `tr_blk_arg_long_dbl_()` (see page 18-63)
- `tr_blk_arg_long_ubits()` (see page 18-66)
- `tr_blk_arg_long_ubits_()` (see page 18-67)
- `tr_blk_arg_short()` (see page 18-68)
- `tr_blk_arg_short_()` (see page 18-68)

- `tr_blk_arg_string()` (see page 18-69)
- `tr_blk_arg_string_()` (see page 18-70)
- `tr_blk_arg_ubits()` (see page 18-71)
- `tr_blk_arg_ubits_()` (see page 18-72)
- `tr_blk_arg_uchar()` (see page 18-73)
- `tr_blk_arg_uchar_()` (see page 18-74)
- `tr_blk_arg_ushort()` (see page 18-75)
- `tr_blk_arg_ushort_()` (see page 18-75)
- `tr_pid()` (see page 18-76)
- `tr_pid_()` (see page 18-77)
- `tr_tid()` (see page 18-78)
- `tr_tid_()` (see page 18-78)
- `tr_thread_id()` (see page 18-79)
- `tr_thread_id_()` (see page 18-80)
- `tr_task_id()` (see page 18-81)
- `tr_task_id_()` (see page 18-81)
- `tr_cpu()` (see page 18-82)
- `tr_cpu_()` (see page 18-83)
- `tr_node()` (see page 18-84)
- `tr_node_()` (see page 18-84)
- `tr_process_name()` (see page 18-85)
- `tr_process_name_()` (see page 18-86)
- `tr_task_name()` (see page 18-86)
- `tr_task_name_()` (see page 18-87)
- `tr_thread_name()` (see page 18-88)
- `tr_thread_name_()` (see page 18-88)

See “Functions” on page 18-9 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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

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

t

data set handle

offset

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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

tr_nargs()

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

SYNTAX

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

PARAMETERS

t

data set handle

RETURN VALUES

Returns the number of arguments associated with the current trace event. In the case of a trace event recorded with `trace_event_string()` or `trace_event_blk()`, it returns the number of four-byte integers that would be required to hold the data.

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

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

t

data set handle

offset

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. In the case of a trace event recorded with `trace_event_string()` or `trace_event_blk()`, it returns the number of four-byte integers that would be required to hold the data.

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

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

t

data set handle

arg_number

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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

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

t

data set handle

arg_number

number of the desired argument

offset

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

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

t

data set handle

arg_number

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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

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

t

data set handle

arg_number

number of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_arg_long()

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

SYNTAX

```
extern long int tr_arg_long (tr_t t,
                             int arg_number);
```

PARAMETERS

t

data set handle

arg_number

number of the desired argument

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

tr_arg_long_()

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

SYNTAX

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

PARAMETERS

t

data set handle

arg_number

number of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

tr_arg_long_dbl()

`tr_arg_long_dbl()` returns the desired double argument of the current trace event.

SYNTAX

```
extern long double tr_arg_long_dbl (tr_t t,
                                    int arg_number);
```

PARAMETERS

t

data set handle

arg_number

number of the desired argument

RETURN VALUES

Returns the desired long 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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_arg_long_dbl_()

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

SYNTAX

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

PARAMETERS

t

data set handle

arg_number

number of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

tr_arg_long_long()

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

SYNTAX

```
extern long long int tr_arg_long_long (tr_t t,
                                       int arg_number);
```

PARAMETERS

t

data set handle

arg_number

number of the desired argument

RETURN VALUES

Returns the desired long long 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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_arg_long_long_()

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

SYNTAX

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

PARAMETERS

t

data set handle

arg_number

number of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

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

t

data set handle

arg_number

number of the desired argument

offset

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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

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

t

data set handle

arg_number

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

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

t

data set handle

arg_number

number of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

tr_arg_long()

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

SYNTAX

```
extern long int tr_arg_long (tr_t t,
                             int arg_number);
```

PARAMETERS

t

data set handle

arg_number

number of the desired argument

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_arg_long_()

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

SYNTAX

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

PARAMETERS

t

data set handle

arg_number

number of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_arg_long_dbl()

`tr_arg_long_dbl()` returns the desired double argument of the current trace event.

SYNTAX

```
extern long double tr_arg_long_dbl (tr_t t,
                                    int arg_number);
```

PARAMETERS

t

data set handle

arg_number

number of the desired argument

RETURN VALUES

Returns the desired long 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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

tr_arg_long_dbl_()

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

SYNTAX

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

PARAMETERS

t

data set handle

arg_number

number of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_arg_long_long()

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

SYNTAX

```
extern long long int tr_arg_long_long (tr_t t,
                                       int arg_number);
```

PARAMETERS

t

data set handle

arg_number

number of the desired argument

RETURN VALUES

Returns the desired long long 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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

tr_argtype()

`tr_argtype()` returns the type of arguments associated with the current event.

SYNTAX

```
extern tr_arg_t tr_argtype (tr_t t);
```

PARAMETERS

t

data set handle

RETURN VALUES

Returns the type of arguments associated with the current event. For events recorded with `trace_event_blk()`, this function returns `int_arg`.

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_arg_t" on page 18-2

tr_argtype_()

`tr_argtype_()` returns the type of arguments associated with the event at the specified offset.

SYNTAX

```
extern tr_arg_t tr_argtype_ (tr_t t, tr_offset_t offset);
```

PARAMETERS

t

data set handle

offset

offset of the trace event

RETURN VALUES

Returns the type of arguments associated with the current (or optionally specified) event. For events recorded with `trace_event_blk()`, this function returns `int_arg`.

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_blk_arg()

`tr_blk_arg()` returns the integer argument at a particular byte offset in argument space of the current trace event.

SYNTAX

```
extern long tr_blk_arg (tr_t t,
                        int byte_offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset 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 byte offset is specified.

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

tr_blk_arg_()

`tr_blk_arg_()` returns the integer argument at a particular byte offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long tr_blk_arg_(tr_t t,
                        int byte_offset,
                        tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_blk_arg_bits()

tr_blk_arg_bits() returns the integer bit field argument of a particular bit size at a particular byte and bit offset offset in argument space of the current trace event.

SYNTAX

```
extern long tr_blk_arg_bits (tr_t t,
                             int byte_offset,
                             int bit_offset,
                             int bit_size);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

bit_offset

bit offset of the desired argument

bit_size

bit size of the desired argument

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_blk_arg_bits_()

`tr_blk_arg_bits_()` returns the integer bit field argument of a particular bit size at a particular byte and bit offset offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long tr_blk_arg_bits_(tr_t t,
                             int byte_offset,
                             int bit_offset,
                             int bit_size,
                             tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

bit_offset

bit offset of the desired argument

bit_size

bit size of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

tr_blk_arg_char()

`tr_blk_arg_char()` returns the character argument at a particular byte offset in argument space of the current trace event.

SYNTAX

```
extern long tr_blk_arg_char (tr_t t,
                             int byte_offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_blk_arg_char_()

`tr_blk_arg_char_()` returns the character argument at a particular byte offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long tr_blk_arg_char_(tr_t t,
                             int byte_offset,
                             tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

offset

offset of the trace event

RETURN VALUES

Returns the desired character argument of the trace event at the specifed offset; returns zero if an invalid offset is specified or an invalid argument byte offset is specified.

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

tr_blk_arg_dbl()

`tr_blk_arg_dbl()` returns the double argument at a particular byte offset in argument space of the current trace event.

SYNTAX

```
extern double tr_blk_arg_dbl (tr_t t,
                              int byte_offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset 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 byte offset is specified.

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_blk_arg_dbl_()

`tr_blk_arg_dbl_()` returns the double argument at a particular byte offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern double tr_blk_arg_dbl_(tr_t t,
                              int byte_offset,
                              tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_blk_arg_flt()

`tr_blk_arg_flt()` returns the float argument at a particular byte offset in argument space of the current trace event.

SYNTAX

```
extern double tr_blk_arg_flt (tr_t t,
                              int byte_offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

tr_blk_arg_flt_()

`tr_blk_arg_flt_()` returns the float argument at a particular byte offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern double tr_blk_arg_flt_(tr_t t,
                              int byte_offset,
                              tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

offset

offset of the trace event

RETURN VALUES

Returns the desired float argument of the trace event at the specifed offset; returns zero if an invalid offset is specified or an invalid argument byte offset is specified.

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_blk_arg_long()

`tr_blk_arg_long()` returns the long integer argument at a particular byte offset in argument space of the current trace event.

SYNTAX

```
extern long tr_blk_arg_long (tr_t t,
                             int byte_offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

tr_blk_arg_long_()

`tr_blk_arg_long_()` returns the long integer argument at a particular byte offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long tr_blk_arg_long_(tr_t t,
                             int byte_offset,
                             tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

tr_blk_arg_long_bits()

`tr_blk_arg_long_bits()` returns the long integer bit field argument of a particular bit size at a particular byte and bit offset offset in argument space of the current trace event.

SYNTAX

```
extern long tr_blk_arg_long_bits (tr_t t,
                                  int byte_offset,
                                  int bit_offset,
                                  int bit_size);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

bit_offset

bit offset of the desired argument

bit_size

bit size of the desired argument

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_blk_arg_long_bits_()

`tr_blk_arg_long_bits_()` returns the long integer bit field argument of a particular bit size at a particular byte and bit offset offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long tr_blk_arg_long_bits_(tr_t t,
                                  int byte_offset,
                                  int bit_offset,
                                  int bit_size,
                                  tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

bit_offset

bit offset of the desired argument

bit_size

bit size of the desired argument

offset

offset of the trace event

RETURN VALUES

Returns the desired long integer bit field argument of the trace event at the specifed offset; returns zero if an invalid offset is specified or an invalid argument byte offset is specified.

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

tr_blk_arg_long_dbl()

`tr_blk_arg_long_dbl()` returns the long double argument at a particular byte offset in argument space of the current trace event.

SYNTAX

```
extern long double tr_blk_arg_long_dbl (tr_t t,
                                        int byte_offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_blk_arg_long_dbl_()

`tr_blk_arg_long_dbl_()` returns the long double argument at a particular byte offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long double tr_blk_arg_long_dbl_(tr_t t,
                                        int byte_offset,
                                        tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

tr_blk_arg_long_long()

`tr_blk_arg_long_long()` returns the long long integer argument at a particular byte offset in argument space of the current trace event.

SYNTAX

```
extern long long tr_blk_arg_long_long (tr_t t,
                                       int byte_offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_blk_arg_long_long_()

`tr_blk_arg_long_long_()` returns the long long integer argument at a particular byte offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long long tr_blk_arg_long_long_(tr_t t,
                                       int byte_offset,
                                       tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_blk_arg_long_ubits()

`tr_blk_arg_long_ubits()` returns the unsigned long integer bit field argument of a particular bit size at a particular byte and bit offset offset in argument space of the current trace event.

SYNTAX

```
extern long tr_blk_arg_long_ubits (tr_t t,
                                   int byte_offset,
                                   int bit_offset,
                                   int bit_size);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

bit_offset

bit offset of the desired argument

bit_size

bit size of the desired argument

RETURN VALUES

Returns the desired unsigned long integer bit field argument of the current trace event; returns zero if an invalid offset is specified or an invalid argument byte offset is specified.

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

tr_blk_arg_long_ubits_()

`tr_blk_arg_long_ubits_()` returns the unsigned long integer bit field argument of a particular bit size at a particular byte and bit offset offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long tr_blk_arg_long_ubits_(tr_t t,
                                   int byte_offset,
                                   int bit_offset,
                                   int bit_size,
                                   tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

bit_offset

bit offset of the desired argument

bit_size

bit size of the desired argument

offset

offset of the trace event

RETURN VALUES

Returns the desired unsigned long integer bit field argument of the trace event at the specifed offset; returns zero if an invalid offset is specified or an invalid argument byte offset is specified.

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_blk_arg_short()

`tr_blk_arg_short()` returns the short integer argument at a particular byte offset in argument space of the current trace event.

SYNTAX

```
extern long tr_blk_arg_short (tr_t t,
                              int byte_offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

tr_blk_arg_short_()

`tr_blk_arg_short_()` returns the short integer argument at a particular byte offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long tr_blk_arg_short_(tr_t t,
                              int byte_offset,
                              tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_blk_arg_string()

`tr_blk_arg_string()` returns a pointer to the null terminated string argument at a particular byte offset in argument space of the current trace event and limited to a particular string size.

SYNTAX

```
extern char *tr_blk_arg_string (tr_t t,
                                int byte_offset,
                                int string_size);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

string_size

the maximum length of the string

RETURN VALUES

Returns the desired string argument of the current trace event; returns NULL if an invalid offset is specified or an invalid argument byte offset is specified.

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

tr_blk_arg_string_()

`tr_blk_arg_string_()` returns a pointer to the null terminated string argument at a particular byte offset in argument space of the trace event at the specified offset and limited to a particular string size.

SYNTAX

```
extern char *tr_blk_arg_string_(tr_t t,
                                int byte_offset,
                                int string_size,
                                tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

string_size

the maximum length of the string

offset

offset of the trace event

RETURN VALUES

Returns the desired string argument of the trace event at the specifed offset; returns zero if an invalid offset is specified or an invalid argument byte offset is specified.

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_blk_arg_ubits()

`tr_blk_arg_ubits()` returns the unsigned integer bit field argument of a particular bit size at a particular byte and bit offset offset in argument space of the current trace event.

SYNTAX

```
extern long tr_blk_arg_ubits (tr_t t,
                              int byte_offset,
                              int bit_offset,
                              int bit_size);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

bit_offset

bit offset of the desired argument

bit_size

bit size of the desired argument

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

tr_blk_arg_ubits_()

`tr_blk_arg_ubits_()` returns the unsigned integer bit field argument of a particular bit size at a particular byte and bit offset offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long tr_blk_arg_ubits_(tr_t t,
                              int byte_offset,
                              int bit_offset,
                              int bit_size,
                              tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

bit_offset

bit offset of the desired argument

bit_size

bit size of the desired argument

offset

offset of the trace event

RETURN VALUES

Returns the desired unsigned integer bit field argument of the trace event at the specifed offset; returns zero if an invalid offset is specified or an invalid argument byte offset is specified.

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_blk_arg_uchar()

`tr_blk_arg_uchar()` returns the unsigned character argument at a particular byte offset in argument space of the current trace event.

SYNTAX

```
extern long tr_blk_arg_uchar (tr_t t,
                              int byte_offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

RETURN VALUES

Returns the desired unsigned character argument of the current trace event; returns zero if an invalid offset is specified or an invalid argument byte offset is specified.

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_blk_arg_uchar_()

`tr_blk_arg_uchar_()` returns the unsigned character argument at a particular byte offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long tr_blk_arg_uchar_(tr_t t,
                              int byte_offset,
                              tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

offset

offset of the trace event

RETURN VALUES

Returns the desired unsigned character argument of the trace event at the specifed offset; returns zero if an invalid offset is specified or an invalid argument byte offset is specified.

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

tr_blk_arg_ushort()

`tr_blk_arg_ushort()` returns the unsigned short integer argument at a particular byte offset in argument space of the current trace event.

SYNTAX

```
extern long tr_blk_arg_ushort (tr_t t,
                               int byte_offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_blk_arg_ushort_()

`tr_blk_arg_ushort_()` returns the unsigned short integer argument at a particular byte offset in argument space of the trace event at the specified offset.

SYNTAX

```
extern long tr_blk_arg_ushort_(tr_t t,
                               int byte_offset,
                               tr_offset_t offset);
```

PARAMETERS

t

data set handle

byte_offset

byte offset of the desired argument

offset

offset of the trace event

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

tr_pid()

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

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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_pid_()

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

SYNTAX

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

PARAMETERS

t

data set handle

offset

offset of the trace event

RETURN VALUES

Returns the 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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

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

t

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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

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

t

data set handle

offset

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

tr_thread_id()

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

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

tr_thread_id_()

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

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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

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

t

data set handle

offset

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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

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. Kernel tracing is not supported on all operating system distributions. See "Kernel Dependencies" on page B-1 for more information.

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

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. Kernel tracing is not supported on all operating system distributions. See “Kernel Dependencies” on page B-1 for more information.

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

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

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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

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

t

data set handle

offset

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

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

t

data set handle

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

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

t

data set handle

offset

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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_offset_t" on page 18-4

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

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 18-30 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

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 18-30 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_offset_t” on page 18-4

Conditions

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

- `tr_cond_create()` (see page 18-91)
- `tr_cond_reset()` (see page 18-92)
- `tr_cond_find()` (see page 18-92)
- `tr_cond_id()` (see page 18-93)
- `tr_cond_id_range()` (see page 18-94)
- `tr_cond_id_clear()` (see page 18-95)
- `tr_cond_cpu()` (see page 18-96)
- `tr_cond_cpu_clear()` (see page 18-97)
- `tr_cond_pid()` (see page 18-98)
- `tr_cond_pid_name()` (see page 18-99)
- `tr_cond_pid_clear()` (see page 18-100)
- `tr_cond_tid()` (see page 18-101)
- `tr_cond_tid_name()` (see page 18-102)
- `tr_cond_tid_clear()` (see page 18-103)
- `tr_cond_node()` (see page 18-104)
- `tr_cond_node_clear()` (see page 18-105)
- `tr_cond_func_or()` (see page 18-106)
- `tr_cond_func_and()` (see page 18-108)
- `tr_cond_func_clear()` (see page 18-110)
- `tr_cond_expr_and()` (see page 18-111)
- `tr_cond_expr_or()` (see page 18-112)
- `tr_cond_not()` (see page 18-113)
- `tr_cond_or()` (see page 18-114)
- `tr_cond_and()` (see page 18-115)
- `tr_cond_copy()` (see page 18-116)
- `tr_cond_name()` (see page 18-118)
- `tr_cond_satisfy()` (see page 18-118)
- `tr_cond_satisfy_()` (see page 18-119)
- `tr_cond_register()` (see page 18-120)
- `tr_cond_offset()` (see page 18-121)

See “Functions” on page 18-9 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

t

data set handle

name

name to subsequently reference newly-created condition; if the name is non-null, the condition may be retrieved via `tr_cond_find()` 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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_find()” on page 18-92

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

t

data set handle

cond

handle of condition to reset

See "Conditions" on page 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_create()" on page 18-91

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

t

data set handle

name

name used to reference the desired condition as defined in `tr_cond_create()`

RETURN VALUES

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

See “Conditions” on page 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_create()” on page 18-91

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

t

data set handle

cond

handle of the condition with which the given trace ID is to be associated

id

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 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_create()" on page 18-91
- "tr_cond_id_range()" on page 18-94
- "tr_cond_id_clear()" on page 18-95

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

t

data set handle

cond

handle of the condition with which the given trace ID range is to be associated

id1

minimum value in the range of trace IDs to be associated with the given condition

id2

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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_id()” on page 18-93
- “tr_cond_id_clear()” on page 18-95

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

t

data set handle

cond

handle of the condition from which all trace ID requirements will be removed

See "Conditions" on page 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_id()" on page 18-93
- "tr_cond_id_range()" on page 18-94

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

t

data set handle

cond

handle of the condition with which to associate the given CPU bias

cpu_bias

CPU bias to apply to the given condition

See "Conditions" on page 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_cpu_clear()” on page 18-97

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

t

data set handle

cond

handle of the condition

See “Conditions” on page 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_cpu()” on page 18-96

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

t

data set handle

cond

handle of the condition

pid

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 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_pid_name()" on page 18-99
- "tr_cond_pid_clear()" on page 18-100

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

t

data set handle

cond

handle of the condition

process_name

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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_pid()" on page 18-98
- "tr_cond_pid_clear()" on page 18-100

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

t

data set handle

cond

handle of the condition

See "Conditions" on page 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_pid()" on page 18-98
- "tr_cond_pid_name()" on page 18-99

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

t

data set handle

cond

handle of the condition

tid

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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_tid_name()” on page 18-102
- “tr_cond_tid_clear()” on page 18-103

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

t

data set handle

cond

handle of the condition

tid_name

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 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_tid()" on page 18-101
- "tr_cond_tid_clear()" on page 18-103

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

t

data set handle

cond

handle of the condition

See “Conditions” on page 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4

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

t

data set handle

cond

handle of the condition

node

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 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_node_clear()" on page 18-105

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

t

data set handle

cond

handle of the condition

See “Conditions” on page 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_node()” on page 18-104

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

t

data set handle

cond

handle of the condition

func

user-callable function to be associated with the given condition

context

user-defined field to be passed to the specified user function

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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_func_t” on page 18-4
- “tr_cond_or()” on page 18-114
- “tr_cond_and()” on page 18-115
- “tr_cond_func_and()” on page 18-108
- “tr_cond_func_clear()” on page 18-110

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

t

data set handle

cond

handle of the condition

func

user-callable function to be associated with the given condition

context

user-defined field to be passed to the specified user function

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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_func_t” on page 18-4
- “tr_cond_or()” on page 18-114
- “tr_cond_and()” on page 18-115
- “tr_cond_func_or()” on page 18-106
- “tr_cond_func_and()” on page 18-108

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

t

data set handle

cond

handle of the condition

See "Conditions" on page 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_func_or()" on page 18-106
- "tr_cond_func_clear()" on page 18-110

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

t

data set handle

cond

handle of the condition

expr

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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_expr_or()” on page 18-112

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

t

data set handle

cond

handle of the condition

expr

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 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_expr_and()" on page 18-111

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

t

data set handle

name

name to reference the newly-created condition; if an existing condition already exists with the specified *name*, it becomes unnamed but remains otherwise unchanged; if *name* is `NULL`, the newly-created condition will be unnamed

cond

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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_or()" on page 18-114
- "tr_cond_and()" on page 18-115

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

t

data set handle

name

name to reference the newly-created condition; if an existing condition already exists with the specified *name*, it becomes unnamed but remains otherwise unchanged; if *name* is `NULL`, the newly-created condition will be unnamed

left

one of two existing conditions either of which must evaluate to `TRUE` for the newly-created condition to evaluate to `TRUE`

right

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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_not()” on page 18-113
- “tr_cond_and()” on page 18-115

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

t

data set handle

name

name to reference the newly-created condition; if an existing condition already exists with the specified *name*, it becomes unnamed but remains otherwise unchanged; if *name* is `NULL`, the newly-created condition will be unnamed

left

one of two existing conditions which must both evaluate to `TRUE` for the newly-created condition to evaluate to `TRUE`

right

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 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_cond_not()" on page 18-113
- "tr_cond_or()" on page 18-114

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

t

data set handle

name

name to reference the newly-created condition; if an existing condition already exists with the specified *name*, it becomes unnamed but remains otherwise unchanged; if *name* is `NULL`, the newly-created condition will be unnamed

cond

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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_cond_or()” on page 18-114
- “tr_cond_and()” on page 18-115

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

t

data set handle

cond

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 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4

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

t

data set handle

cond

handle of the condition

RETURN VALUES

Returns `TRUE` if the current event satisfies the specified condition; returns `FALSE` otherwise.

See “Conditions” on page 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4

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

t

data set handle

cond

handle of the condition

offset

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 18-90 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4
- "tr_offset_t" on page 18-4

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

t

data set handle

cond

handle of condition to register

ADDITIONAL INFORMATION

This is the implementation of NightTrace "profiles" 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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_activate()” on page 18-133
- “Profile References” on page 16-193

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

t

data set handle

cond

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 18-90 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4
- “tr_offset_t” on page 18-4

State-oriented Interfaces

The functions that deal with the creation, configuration, and activation of states are:

- `tr_state_create()` (see page 18-122)
- `tr_state_find()` (see page 18-123)
- `tr_state_name()` (see page 18-124)
- `tr_state_start_id()` (see page 18-125)
- `tr_state_start_id_range()` (see page 18-126)
- `tr_state_start_id_clear()` (see page 18-127)
- `tr_state_end_id()` (see page 18-127)
- `tr_state_end_id_range()` (see page 18-128)
- `tr_state_end_id_clear()` (see page 18-129)
- `tr_state_start_cond()` (see page 18-130)
- `tr_state_start_cond_clear()` (see page 18-131)
- `tr_state_end_cond()` (see page 18-131)
- `tr_state_end_cond_clear()` (see page 18-132)
- `tr_activate()` (see page 18-133)
- `tr_state_info()` (see page 18-134)
- `tr_state_info_()` (see page 18-135)
- `tr_state_active()` (see page 18-136)
- `tr_state_active_()` (see page 18-137)

See "Functions" on page 18-9 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

t

data set handle

name

name to reference the newly-created state; if an existing state already exists with the specified *name*, it becomes unnamed but remains otherwise unchanged; if *name* 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 18-122 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8

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

t

data set handle

name

name used to reference the desired state as defined in `tr_state_create()`

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 18-122 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_state_t" on page 18-7
- "tr_state_create()" on page 18-122

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

t

data set handle

state

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 18-122 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_state_t” on page 18-7

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

t

data set handle

state

handle of the state

id

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 18-122 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_state_t” on page 18-7

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

t

data set handle

state

handle of the state

id1

minimum value in the range of trace IDs to be associated with the given state

id2

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 18-122 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_state_t" on page 18-7

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

t

data set handle

state

handle of the state

See “State-oriented Interfaces” on page 18-122 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_state_t” on page 18-7

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

t

data set handle

state

handle of the state

id

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 18-122 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_state_t" on page 18-7

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

t

data set handle

state

handle of the state

id1

minimum value in the range of trace IDs to be associated with the given state

id2

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 18-122 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_state_t” on page 18-7

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

t

data set handle

state

handle of the state

See “State-oriented Interfaces” on page 18-122 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_state_t” on page 18-7

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

t

data set handle

state

handle of the state

cond

handle of the condition to associate with the start of the specified state

See "State-oriented Interfaces" on page 18-122 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_state_t" on page 18-7
- "tr_cond_t" on page 18-4

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

t

data set handle

state

handle of the state

See “State-oriented Interfaces” on page 18-122 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_state_t” on page 18-7

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

t

data set handle

state

handle of the state

cond

handle of the condition to associate with the end of the specified state

See "State-oriented Interfaces" on page 18-122 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_state_t" on page 18-7
- "tr_cond_t" on page 18-4

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

t

data set handle

state

handle of the state

See "State-oriented Interfaces" on page 18-122 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_state_t" on page 18-7

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 18-122 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_state_active()” on page 18-136

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

t

data set handle

state

handle of the state

info

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 18-6).

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 18-122 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_state_t" on page 18-7
- "tr_state_info_t" on page 18-6

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

t

data set handle

state

handle of the state

info

pointer to a structure which will contain the current values associated with the given state at the specified offset

offset

offset of the specifed state

RETURN VALUES

The return values are contained in the `tr_state_info_t` structure (see “tr_state_info_t” on page 18-6).

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 18-122 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_state_t" on page 18-7
- "tr_state_info_t" on page 18-6
- "tr_offset_t" on page 18-4

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

t

data set handle

state

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 18-122 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_state_t" on page 18-7

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

t

data set handle

state

handle of the state

offset

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 18-122 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_state_t” on page 18-7
- “tr_offset_t” on page 18-4

Output Function

The function dealing with the output of trace data is:

- `tr_copy_input()` (see page 18-138)
- `tr_copy_input_range()` (see page 18-139)

See "Functions" on page 18-9 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

t

data set handle

output_file

pathname of the output file

cond

handle of the condition

mode

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 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_cond_t" on page 18-4

tr_copy_input_range()

`tr_copy_input_range()` copies all the events in the data set whose offsets lie in the range specified.

SYNTAX

```
extern int tr_copy_input_range (tr_t t,
                                char * output_file,
                                int mode);
                                int start);
                                int end);
```

PARAMETERS

t

data set handle

output_file

pathname of the output file

mode

parameter passed to the system call invoked to open/create the specified output file

start

start of the range

end

end of the range

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 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4

String Table Functions

The following functions are provided to create, manage, and search NightTrace string tables:

- `tr_get_string()` (see page 18-140)
- `tr_get_item()` (see page 18-141)
- `tr_create_table()` (see page 18-142)
- `tr_append_table()` (see page 18-143)

See "Functions" on page 18-9 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

t

data set handle

table_name

name of the string table

item

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 18-140 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “String Tables” on page 7-15

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

t

data set handle

table_name

name of the table to search for the specified string

value

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 18-140 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “String Tables” on page 7-15

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

t

data set handle

table_name

name to subsequently reference the newly-created table

default_value

string to associate with integer values that are not explicitly referenced in the table

list

pointer to a list of string table entries

count

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 18-140 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8

- “tr_string_node_t” on page 18-7
- “String Tables” on page 7-15

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

t

data set handle

table_name

name of the table to modify

value

character string to assign to the given item number

item

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 18-140 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "String Tables" on page 7-15

Callback Interfaces

The following functions deal with the callback capabilities of the NightTrace Analysis Application Programming Interface:

- `tr_iterate()` (see page 18-145)
- `tr_halt()` (see page 18-146)
- `tr_cancel_cb()` (see page 18-146)
- `tr_cond_cb()` (see page 18-147)
- `tr_state_cb()` (see page 18-148)

See “Functions” on page 18-9 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 18-145 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_cb()” on page 18-147
- “tr_state_cb()” on page 18-148
- “tr_open_stream()” on page 18-19

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 18-145 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 18-8
- "tr_iterate()" on page 18-145

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 18-145 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cb_t” on page 18-3

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

t

data set handle

cond

handle of the condition that must be satisfied in order for the callback function to be called

func

function to be called if the given condition is satisfied for a particular event

context

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 18-145 for related functions.

See “Functions” on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_cond_t” on page 18-4

- "tr_cond_cb_func_t" on page 18-3
- "tr_cb_t" on page 18-3

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

t

data set handle

state

handle of the state

action

specifies the manner in which the given function will be called (see "tr_state_action_t" on page 18-5)

func

function which will be iteratively invoked for every event that affects the given state in the specified manner

context

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 18-145 for related functions.

See "Functions" on page 18-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 18-8
- “tr_state_t” on page 18-7
- “tr_state_action_t” on page 18-5
- “tr_state_cb_func_t” on page 18-5
- “tr_cb_t” on page 18-3

A
NightStar Licensing

NightStar RT uses the NightStar License Manager (NSLM) to control access to the NightStar RT tools.

License installation requires a licence key provided by Concurrent (see “License Keys” on page A-1).The NightStar RT tools request a licence (see “License Requests” on page A-2) from a license server (see “License Server” on page A-2).

Two license modes are available, fixed and floating, depending on which product option you purchased. Fixed licenses can only be served to NightStar RT users from the local system. Floating licenses may be served to any NightStar RT user on any system on a network.

Tools are licensed per system, per concurrent user. A single license is shared among any or all of the NightStar RT tools for a particular user on a particular system. The intent is to allow *n* developers to fully utilize all the tools at the same time while only requiring *n* licenses. When operating the tools in remote mode, where a tool is launched on a local system but is interacting with a remote system, licenses are required only from the host system.

You can obtain a license report which lists all licenses installed on the local system, current usage, and expiration date for demo licenses (see “License Reports” on page A-3).

The default configuration includes a strict firewall which interferes with floating licenses. See “Firewall Configuration for Floating Licenses” on page A-3 for information on handling such configurations.

See “License Support” on page A-4 for information on contacting Concurrent for additional assistance with licensing issues.

License Keys

Licenses are granted to specific systems to be served to either local or remote clients, depending on the license model, fixed or floating.

License installation requires a license key provided by Concurrent. To obtain a license key, you must provide your system identification code. The system identification code is generated by the **`nslm_admin`** utility:

```
nslm_admin --code
```

System identification codes are dependent on system configurations. Reinstalling Linux on a system or replacing network devices may require you to obtain new license keys.

To obtain a license key, use the following URL:

http://www.ccur.com/NightStarRTKeys

Provide the requested information, including the system identification code. Your license key will be immediately emailed to you.

Install the license key using the following command:

```
nslm_admin --install=xxxx-xxxx-xxxx-xxxx-xxxx
```

where *xxxx-xxxx-xxxx-xxxx-xxxx* is the key included in the license acknowledgment email.

License Requests

By default, the NightStar RT tools request a license from the local system. If no licenses are available, they broadcast a license request on the local subnet associated with the system's hostname.

You can control the license requests for an entire system using the **/etc/nslm.config** configuration file.

By default, the **/etc/nslm.config** file contains a line similar to the following:

```
:server @default
```

The argument @default may be changed to a colon-separated list of system names, system IP addresses, or broadcast IP addresses. Licenses will be requested from each of the entities found in the list, until a license is granted or all entries in the list are exhausted.

For example, the following setting prevents broadcast requests for licenses, by only specifying the local system:

```
:server localhost
```

The following setting requests a license from **server1**, then **server2**, and then a broadcast request if those fail to serve a license:

```
:server server1:server2:192.168.1.0
```

Similarly, you can control the license requests for individual invocations of the tools using the **NSLM_SERVER** environment variable. If set, it must contain a colon-separated list of system names, system IP addresses, or broadcast IP addresses as described above. Use of the **NSLM_SERVER** environment variable takes precedence over settings defined in **/etc/nslm.config**.

License Server

The NSLM license server is automatically installed and configured to run when you install NightStar RT.

The **nslm** service is automatically activated for run levels 2, 3, 4, and 5. You can check on these settings by issuing the following command:

```
/sbin/chkconfig --list nslm
```

In rare instances, you may need to restart the license server via the following command:

```
/sbin/service nslm restart
```

See **nslm(1)** for more information.

License Reports

A license report can be obtained using the **nslm_admin** utility.

```
nslm_admin --list
```

lists all licenses installed on the local system, current usage, and expiration date (for demo licenses). Use of the **--verbose** option also lists individual clients to which licenses are currently granted.

Adding the **--broadcast** option will list this information for all servers that respond to a broadcast request on the local subnet associated with the system's hostname.

See **nslm_admin(1)** for more options and information.

Firewall Configuration for Floating Licenses

RedHawk does not support a firewall configuration by default, because iptables support is disabled. However, it is possible to build a custom kernel with iptables support enabled. If that is done, and floating licenses are used, the iptables firewall rules must be configured to allow the license requests and responses to pass.

If the system with iptables support and firewall rules is serving licenses, then the firewall rules must be arranged to allow license requests on UDP port 25517 and TCP port 25517 from any systems that will make license requests. For example, in a simple firewall, rules like the following, inserted before any DROP or REJECT rules, might work:

```
iptables -A INPUT -p udp -m udp -s subnet/mask --dport 25517 -j ACCEPT
iptables -A INPUT -p tcp -m tcp -s subnet/mask --dport 25517 -j ACCEPT
```

If the system with iptables support and firewall rules is running NightStar RT tools and receiving floating licenses, then the firewall rules must be arranged to allow license responses on UDP port 25517 from any system serving licenses. For example, in a simple firewall, rules like the following, inserted before any DROP or REJECT rules, might work:

```
iptables -A INPUT -p udp -m udp -s subnet/mask --sport 25517 -j ACCEPT
```

License Support

For additional aid with licensing issues, contact the Concurrent Software Support Center at our toll free number 1-800-245-6453. For calls outside the continental United States, the number is 1-954-283-1822. The Software Support Center operates Monday through Friday from 8 a.m. to 5 p.m., Eastern Standard Time.

You may also submit a request for assistance at any time by using the Concurrent Computer Corporation web site at **http://www.ccur.com/isd_support_contact.asp** or by sending an email to **support@ccur.com**.

B
Kernel Dependencies

Concurrent's RedHawk kernel provides features and performance gains that are critical for the optimal operation of the NightStar RT tools.

The NightStar RT tools can operate in a host-only mode on Red Hat systems without Concurrent's RedHawk kernel, cross-targeting to RedHawk systems.

Additionally, the NightStar RT tools can function on Red Hat systems without the RedHawk kernel, but will lack the numerous advantages afforded by running with it.

The following sections describe the additional functionality and capabilities of the NightStar RT tools when running Concurrent's RedHawk kernel

Advantages for NightView

The following advantages are afforded NightView when Concurrent's RedHawk kernel is running:

- Application speed conditions

  Provides "execution-speed" patches, conditions, and ignore counts.

- Signal handling

  Allows NightView to pass signals directly to a particular process, avoiding context switching.

Advantages for NightTrace

The following advantage is afforded NightTrace when Concurrent's RedHawk tracing kernel is running:

- Kernel tracing

  Users of NightTrace gain the ability to obtain kernel trace data and combine that with user trace data. Kernel tracing is an incredibly powerful feature that not only provides insight into the operating system kernel but also provides useful information relating to the execution of user applications.

  The RedHawk kernel is provided in three flavors:

  - Tracing

- Debug
- Plain

The Tracing and Debug flavors provide the features required for NightTrace kernel tracing. These kernels can be selected at boot-time from the boot-loader menu.

Advantages for NightProbe

The following advantages are afforded NightProbe when Concurrent's RedHawk a RedHawk or SLERT kernel is running:

- Minimal intrusion

  Allows NightProbe to read and write variables without stopping the process for each sample or write operation.

- Sampling performance

  Allows NightProbe to use direct memory fetches for data sampling (as opposed to programmed I/O) which is important for high-rate data acquisition.

- Concurrent debugging/probing

  Allows NightProbe to probe programs already under the control of a debugger or another NightProbe session.

- PCI Device probing

  Allows NightProbe to probe PCI device memory via the Base Address Register (BAR) file system.

Advantages for NightTune

The following advantage is afforded NightTune when Concurrent's RedHawk a RedHawk or SLERT kernel is running:

- Context switch rate

  Allows NightTune user to display the context switch counts per CPU instead of for the overall system.

- CPU shielding

  Individual CPUs can be shielded from interrupts and processes allowing CPUs to be dedicated solely to specific interrupts and processes that are bound to the CPU.

- CPU sibling interference

  Individual CPUs can be marked down to avoid interfering with hyperthreaded sibling CPUs and dual-core sibling CPUs. Hyperthreaded CPUs share all the resources of their sibling CPU. Dual-core CPUs share the CPU cache and a path to memory with their sibling CPU.

- Detailed memory information

  Detailed process memory descriptions include the residency and lock state of any page in a process, and their association with physical memory pools for NUMA systems.

Frequency Based Scheduler

The Frequency Based Scheduler is only available on RedHawk systems from Concurrent Computer Corporation. It is required for all NightSim usage.

NightSim is only included in NightStar distributions intended for use on RedHawk systems.

PCI Bar File System

The PCI Bar File System is only available with the RedHawk kernel from Concurrent Computer Corporation and SLERT versions 1.0-1.6 kernel from Novell.

On other systems, PCI Device probing will be disabled within NightProbe.

C
Privileged Access

Some features of NightTrace require either root access or privileged access as described below.

This chapter provides an overview of the capabilities mechanism support by some operating systems.

The following operating system kernels support the capabilities mechanism:

- RedHawk Linux (all versions)
- SUSE Linux Enterprise Real Time (versions 1.0-1.6 only)

Capabilities

The following capabilities may be required when using NightTrace:

- `CAP_SYS_NICE`

  If you wish to run the **`ntraceud`** daemon with a real-time scheduling policy and priority, you must have this capability. For example:

  ```
  ntraceud --policy=fifo --priority=50 data-file
  ```

- `CAP_IPC_LOCK`

  If you wish to run the **`ntraceud`** daemon and force shared pages between the user application and the daemon to be locked in memory, you must have this capability. Similarly, this capability is required if you specify page locking when configuring a daemon via the API. For example:

  ```
  ntraceud --lock data-file
  ```

  or

  ```
  ntconfig_t config;
  trace_default_config(&config);
  config.ntc_lock_pages = ntp_lock;
  config.ntc_daemon_preferred = false;
  trace_begin("data-file",&config);
  ```

Linux provides a means to grant otherwise unprivileged users the authority to perform certain privileged operations. The Pluggable Authentication Module (see **`pam_capability(8)`**) is used to manage sets of capabilities, called *roles*, required for various activities.

Linux systems should be configured with an `ntraceuser` role which provides the `CAP_SYS_NICE` and `CAP_IPC_LOCK` capabilities.

Edit **`/etc/security/capability.conf`** and define the `ntraceuser` role (if it is not already defined) in the "ROLES" section:

```
role ntraceuser  CAP_SYS_NICE CAP_IPC_LOCK
```

Additionally, for each NightTrace user on the target system, add the following line at the end of the file:

```
user  username     ntraceuser
```

where *username* is the login name of the user.

If the user requires capabilities not defined in the `ntraceuser` role, add a new role which contains `ntraceuser` and the additional capabilities needed, and substitute the new role name for `ntraceuser` in the text above.

In addition to registering your login name in **`/etc/security/capability.conf`**, certain files under the **`/etc/pam.d`** directory must also be configured to allow capabilities to be activated.

To activate capabilities, add the following line to the end of selected files in **`/etc/pam.d`** if it is not already present:

```
session required pam_capability.so
```

The list of files to modify is dependent on the list of methods that will be used to access the system. The following table presents a recommended configuration that will grant capabilities to users of the services most commonly employed in accessing a system.

Table C-1. Recommended /etc/pam.d Configuration

/etc/pam.d File	Affected Services	Comment
`remote`	telnet rlogin rsh (when used w/o a command)	Depending on your system, the **`remote`** file may not exist. Do not create the **`remote`** file, but edit it if it is present.
`login`	local login (e.g. console) telnet* rlogin* rsh* (when used w/o a command)	*On some versions of Linux, the presence of the **`remote`** file limits the scope of the **`login`** file to local logins. In such cases, the other services listed here with **`login`** are then affected solely by the **`remote`** configuration file.
`rsh`	rsh (when used with a command)	e.g. **`rsh system_name a.out`**
`sshd`	ssh	You must also edit **`/etc/ssh/sshd_config`** and ensure that the following line is present: **`UsePrivilegeSeparation no`**
`gdm`	gnome sessions	
`kde`	kde sessions	

If you modify **/etc/pam.d/sshd** or **/etc/ssh/sshd_config**, you must restart the **sshd** service for the changes to take effect:

```
service sshd restart
bash /etc/init.d/sshd restart
```

In order for the above changes to take effect, the user must log off and log back onto the target system.

NOTE

To verify that you have been granted capabilities, issue the following command:

```
/usr/sbin/getpcaps $$
/sbin/getpcaps $$
```

The output from that command will list the roles currently assigned to you.

D
NightTrace Logging API Examples

This chapter provides several examples using the NightTrace Logging API.

Single Threaded C Example

This example uses demonstrates a minimalist approach to tracing, foregoing any error checking and logging very simple events.

```
#include <ntrace.h>

main()
{
   volatile double x = 0.0;
   int i,j;

   trace_begin ("data",0);

   for (j=0; j<100; ++j) {
      trace_event (1);
      for (i=0; i<1000; ++i) {
         x = x * x;
      }
      trace_event (2);
   }
}
```

The call to `trace_begin`() initializes tracing with default parameters.

We call `trace_event`() with different event identifiers immediately before and after our application's workload, represented by the inner loop.

The following commands could be used to compile, link, and execute the application using command-line daemon execution:

```
$ cc -g file.c -lntrace
$ ntraceud data; ./a.out; ntraceud -q data
```

Using the command line summary option to ntrace, print a summary of each execution of the outer loop:

```
$ ntrace --summary=st:1-2 data
======================================================================
Summary: States starting with event 1, ending with event 2:
```

```
Condition Summary Results
=========================

Number of matching events found:       100
Average gap between matching events:   0.000000184

Maximum gap between matching events:   0.000000391
Maximum gap event offset:              15

Minimum gap between matching events:   0.000000165
Minimum gap event offset:              17
```

Multi-Threaded C++ Example

This example demonstrates using NightTrace event logging from multiple threads.

```
#include <stdio.h>
#include <stdlib.h>
#include <ntrace.h>
#include <time.h>

#define Start 100
#define End 200

volatile int done = 0;

int work (int input)
{
   // do something
   return input;
}

void *
thread_a (void * ptr)
{
   int i = 0;
   int result;
   trace_set_thread_name ("romeo");
   struct timespec ts = { 0, 20000000};
   while (!done) {
      trace_event_arg (Start, i);
      result = work(i++);
      trace_event_arg(End, result);
      nanosleep(&ts,0);
   }
}

void *
thread_b (void * ptr)
{
   int i=9999999;
   int result;
   trace_set_thread_name ("juliet");
   struct timespec ts = { 0, 20000000};
   while (!done) {
      trace_event_arg (Start, i);
      result = work(i--);
      trace_event_arg(End, result);
      nanosleep(&ts,0);
   }
}

int
main (int argc, char * argv[])
{
   pthread_t thread;
   pthread_attr_t attr;
   int status;

   status = trace_begin ("data",NULL);
   switch (status) {
   case NTLISTEN:
      printf ("No daemon is listening -- "
              "proceeding in case one shows up\n");
```

```
            break;
         case NTNOERROR:
            break;
         default:
            printf ("An error occurred during ntrace initialization (%d)\n",
                    status);
            exit(1);
         }

         pthread_attr_init(&attr);
         pthread_create (&thread, &attr, thread_a, NULL);

         pthread_attr_init(&attr);
         pthread_create (&thread, &attr, thread_b, NULL);
         sleep(1);

         done = 1;
      }
```

The call to `trace_begin`() initializes tracing with default parameters.

Immediately within the thread routines, each thread identifies itself with a symbolic name via a call to `trace_set_thread_name()`. If these calls were not made, the threads would be automatically named by NightTrace using the thread's internal **`gettid(2)`** integer value.

The following commands could be used to compile, link, and execute the application using command-line daemon execution:

```
$ cc -g file.c -lntrace_thr -lpthread
$ ntraceud data; ./a.out; ntraceud -q data
```

NOTE

Note the use of the thread-aware version of the NightTrace logging API library, **`-lntrace_thr`**. This is required for use with multi-threaded programs if you want to be able to distinguish between individual threads in trace events. See "Threads and Logging" on page 2-33 for more information).

The following command invokes **`ntrace`** to graphically view the events. A customized page is automatically built which distinguishes events between the two threads: `romeo` and `juliet`:

```
$ ntrace data
```

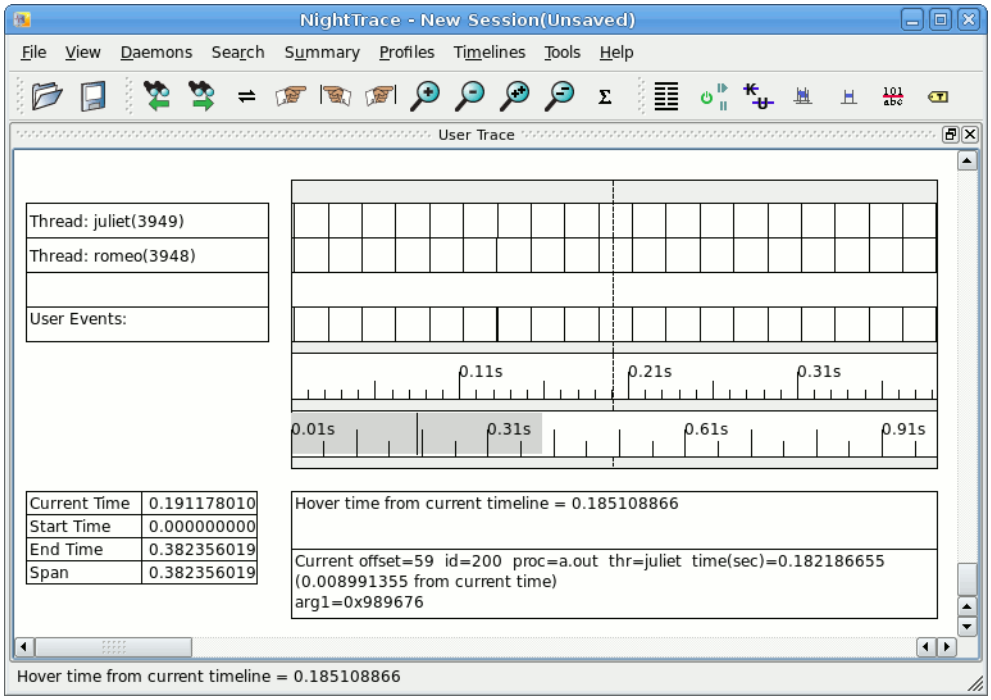


Figure C-1. Automatically Generated Data Display Page

Fortran Example

This example uses demonstrates a simple Fortran program logging a trace event.

```
      program ftrace

      include "/usr/include/ntrace_.h"

      integer void

      void = trace_begin("data",0)

      do 10 i=1,10
         void = trace_event_arg(1,i)
10    continue

      void = trace_end()

      end
```

The call to `trace_start`() initializes tracing with default parameters.

We call `trace_event_arg`() with the loop iterator for each iteration.

The following commands could be used to compile, link, and execute the application using command-line daemon execution:

```
$ g77 -g file.c -lntrace
$ ntraceud data; ./a.out; ntraceud -q data
```

Using the command line listing option to ntrace, we see the values of the iterator as event points are logged:

```
$ ntrace --listing data
0: cpu=?? 1  pid=a.out     thr=main        time=0.000000000s arg1=0x1
1: cpu=?? 1  pid=a.out    thr=main        time=0.000002481s arg1=0x2
2: cpu=?? 1  pid=a.out     thr=main        time=0.000003103s arg1=0x3
3: cpu=?? 1  pid=a.out     thr=main        time=0.000003536s arg1=0x4
4: cpu=?? 1  pid=a.out     thr=main        time=0.000003976s arg1=0x5
5: cpu=?? 1  pid=a.out     thr=main        time=0.000004386s arg1=0x6
6: cpu=?? 1  pid=a.out     thr=main        time=0.000004882s arg1=0x7
7: cpu=?? 1  pid=a.out     thr=main        time=0.000005302s arg1=0x8
8: cpu=?? 1  pid=a.out     thr=main        time=0.000005820s arg1=0x9
9: cpu=?? 1  pid=a.out     thr=main        time=0.000006294s arg1=0xa
...
```

Simple Java Example

This example demonstrates a minimalist approach to tracing in Java, foregoing any error checking and logging very simple events.

```
import ntrace.logging.Trace;

class SimpleTracing {
   public static void main (String args[]) {
      Trace.begin("data");
      for (int j=0; j<10; j++) {
         Trace.event(1,j);
      }
      System.out.println("That was easy!");
   }
}
```

The call to `Trace.begin()` initializes tracing with default parameters.

In the code above, we call `Trace.event()` with different trace argument values inside the loop, but always with the trace event ID of 1.

The following commands could be used to build and execute the application:

```
$ javac -classpath /usr/lib:. SimpleTracing.java
$ ntraceud data
$ java -classpath /usr/lib:. SimpleTracing
That was easy!
$ ntraceud -q data
```

NOTE:

Some versions of java require you to explicitly place **/usr/lib** in your **LD_LIBRARY_PATH** environment variable. If you get an error when invoking the java class that mentions java.lang.UnsatisfiedLinkError, then try setting **LD_LIBRARY_PATH** to include **/usr/lib**.

Use the command line **--summary** option to **ntrace** to summarize the logged events:

```
$ ntrace --summary=ev:1 data

======================================================================
Summary: Occurrences of event 1

Condition Summary Results
=========================

Number of matching events found:       10
Average gap between matching events:   0.000002778

Maximum gap between matching events:   0.000018644
Maximum gap event offset:              5

Minimum gap between matching events:   0.000000975
Minimum gap event offset:              9
```

Multi-Threaded Java Example

This example demonstrates tracing in a multi-threaded Java program.

```
import ntrace.logging.Trace;

class ThreadedTracing {

   public static class MyThread extends Thread {
      public MyThread (String name) {
         super(name);
      }
      public void run () {
         Trace.setThreadName(getName());
         for (int i=0; i<10; ++i) {
            int nap = (int)(Math.random()*100);
            Trace.event(100,nap);
            try {
               Thread.sleep(nap);
            } catch (InterruptedException e) {}
            Trace.event(101);
         }
      }
   }

   public static void main (String args[]) {
      Trace.begin("data");
      new MyThread("one").start();
      new MyThread("two").start();
      System.out.println("That was still easy!");
   }
}
```

The call to `Trace.begin()` initializes tracing with default parameters.

In the code above, we create a user-defined thread class called `MyThread`.

The first thing we do in the body of the thread is tell the NightTrace API the name of our thread by calling `Trace.setThreadName()`. This is a convenience, but important, so we can subsequently determine in NightTrace which thread logged a specific trace point by using its name, instead of an integer thread ID which changes from run to run and cannot be predicted.

In each iteration of the loop in the `run()` routine, each thread will sleep for a random amount of time, surrounded by a pair of trace events with ID values of 100, and 101, respectively. When logging event ID 100, we also have chosen to log an integer argument, which represents the amount of time we will request to sleep.

The `main` java routine creates two instances of `MyThread` and names them "one" and "two".

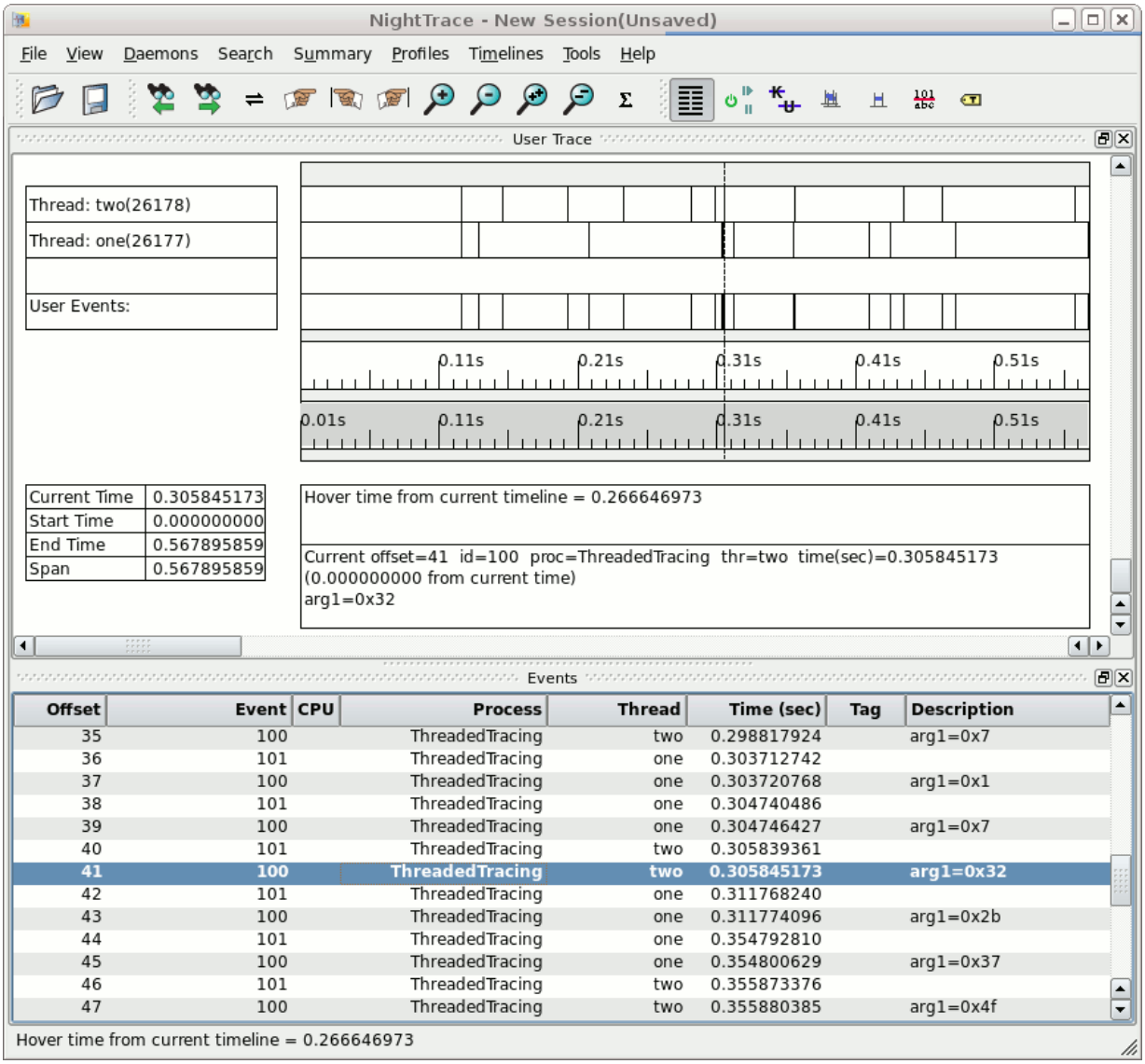
The following commands could be used to build and execute the application:

```
$ javac -classpath /usr/lib:. ThreadedTracing.java
$ ntraceud data
$ java -classpath /usr/lib:. ThreadedTracing
That was still easy!
$ ntraceud -q data
```

NOTE

Some versions of Java require you to explicitly place **`/usr/lib`** in your **`LD_LIBRARY_PATH`** environment variable. If you get an error when invoking the java class that mentions `java.lang.UnsatisfiedLinkError`, then try setting **`LD_LIBRARY_PATH`** to include **`/usr/lib`**.

If we now invoke the NightTrace analysis program, **`ntrace(1)`**, we will see a display which breaks down the trace events on a per-thread basis, as seen in the rows in the center of the figure below.



Rare Occurrence Example

This example uses demonstrates how one might use buffer-wrap mode to catch a rare occurrence of bug.

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

void
incredibly_rare_event (void)
{
   trace_event(2);
   time_t t = time(0);
   printf ("a.out: Badness occurred at %s", asctime(localtime(&t)));
   trace_flush();
}

main()
{
   volatile double x = 0.0;
   int j;
   unsigned i = 0;

   trace_begin ("data",0);
   for (;;) {
      trace_event_arg (1,i);
      for (j=0; j<100; ++j) x = x * x;
      if ((++i % 10000000) == 0) {
         incredibly_rare_event();
      }
   }
}
```

The call to `trace_begin`() initializes tracing with default parameters.

We call `trace_event_arg`() with the loop iterator for each iteration of the outer loop to simulate logging useful data.

When the process detects something has gone wrong, it logs a new trace event and then flushes the trace buffers with a call to `trace_flush`().

The following commands could be used to compile, link, and execute the application using command-line daemon execution:

```
$ cc -g file.c -lntrace
$ ntraceud --bufferwrap data
$ ./a.out &
a.out: Badness occurred at Fri Oct  7 18:00:26 2005
a.out: Badness occurred at Fri Oct  7 23:12:55 2005
$ ntraceud --quit-now data
$ jobs
[1] + Running            a.out
a.out: Badness occurred at Sat Oct  8 02:45:01 2005
```

```
a.out: Badness occurred at Sat Oct  8 08:21:17 2005
```

The program continues to execute despite the detection of the condition, but on each detection, the history of events that were still in the trace shared memory buffers are written to the output file.

The latter invocation of **`ntraceud`** to stop the daemon, indicates is should not wait for the logging application to complete.

We can now analyze the data from the two occurrences of the problematic event.

Alternatively, we could have started the program without an **`ntraceud`** daemon running, and subsequently used the **`ntrace`**, the NightTrace GUI to start a daemon, and immediately analyze the trace data as more data is being collected.

E
NightTrace Analysis API Examples

The following programs are given as examples of how to use the NightTrace Analysis Application Programming Interface (see “Using the NightTrace Analysis API” on page 18-1).

NOTE

The source files for these programs are installed in **`/usr/lib/NightTrace/examples`**.

- **`list`** (see “list” on page E-2)

  This program simply lists each NightTrace event using a simple main loop to position to the next event.

- **`search`** (see “search” on page E-4)

  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 E-6)

  This program illustrates how to monitor a certain condition in real-time and then act upon it accordingly.

- **`ptime`** (see “ptime” on page E-9)

  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 E-12)

  This program contains a collection of code segments which might be useful for reference.

- **`detect`** (see “detect” on page E-23)

  This program monitors live kernel trace data looking for a user-specified event in the form of a NightTrace expression.

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 "NightTrace Analysis API Examples" on page E-1 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 "NightTrace allows you to use expressions to aid in the analysis of trace data." on page 16-1) 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 "NightTrace Analysis API Examples" on page E-1 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*.

This test program make use of kernel tracing which is not available on all operating system distributions. See "Kernel Dependencies" on page B-1 for more information.

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. See "Consumer" on page 9-10 for more information.

See "NightTrace Analysis API Examples" on page E-1 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 "NightTrace Analysis API Examples" on page E-1 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 "NightTrace Analysis API Examples" on page E-1 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.

tr_t t;

static char buffer[128];
static char * _c;
static FILE * input;

#define get_line(x) \
   write (1, x, sizeof(x)); \
   _c = fgets(buffer,sizeof(buffer),input); \
   _c[strlen(_c)-1] = '\0'

static
void
print (tr_offset_t offset)
{
   int i;
```

```
   double time = tr_time(t);
   char * process = tr_process_name(t);

   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;
      }
```

detect

Usage

`./detect` *expression*

This program monitors live kernel trace data looking for a user-specified event in the form of a NightTrace expression. When the event is detected, it writes out a kernel trace data file which contains the detected event as well as 500 events previous to it. It then terminates.

This program illustrates how to monitor a certain condition in real-time and then save trace data prior to and including the event when the condition was detected.

This would be useful in order to collect kernel trace data continually until some complex event occurs - then to save the relevant kernel data for later analysis.

This program may be invoked with the following command:

```
ntracekd --stream /tmp/handle | ./detect "process_name==\"ntracekd\""
```

or it can be launched from the NightTrace GUI as part of a streaming kernel daemon definition. See “Consumer” on page 9-10 for more information.

In this case, the expression provided instructs the program to look for the first kernel event associated with the daemon that is collecting the kernel data and sending it to our **`./detect`** program. This example is used simply for demonstration - it is not very interesting in and of itself.

After executing has stopped, a kernel trace data file called **`copy_current_input.data`** has been written to the current working directory. You can invoke **`ntrace`** on that data file to view the 500 events just prior to the first **`ntracekd`** event:

```
ntrace copy_current_input.data
```

NOTE

There may be fewer than 500 events saved since we may encounter **`ntracekd`** almost immediately.

See “NightTrace Analysis API Examples” on page E-1 for other programs demonstrating the capabilities of the NightTrace Analysis Application Programming Interface.

detect.c

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <ntrace_analysis.h>

// This program detects the first event where the expression is true
// and saves the desired number of events to the output file.


static char* detect_usage =
"Usage: \n"
"\n"
"      ntracekd --stream output | ./detect 500 \"NightTrace Expression\" \n"
"\n"
"           This will detect the first event where the condition is met  \n"
"           and copy the last 500 events prior to that event to the output \n"
"           file.  Tracing will be stopped at that point.  \n"
"\n"
"      ntracekd --stream output | ./detect --bracket 500 \"NightTrace Expression\"
\n"
"\n"
"          This will detect the first event where the condition is met  \n"
"          and copy the 500 events prior to and after that event to the \n"
"          output file.  Tracing will be stopped at that point.  \n"
"\n"
;


// IMPORTANT: stdin is assumed to be the output of ntracekd (or detect was
// launched from the NightTrace GUI which set stdin to daemon output).

// Callbacks
static void copy_input_range_cb
  (tr_t        t,
   tr_state_t  state,
   tr_offset_t offset,
   int         occurrence,
   void      * context,
   int       * disable);

static void copy_current_input_cb
  (tr_t        t,
   tr_state_t  state,
   tr_offset_t offset,
   int         occurrence,
   void      * context,
   int       * disable);


static int range = 0;

int
main (int argc, char * argv[])
```

```
{
   tr_t t;
   tr_cond_t user;
   tr_cond_t start;
   tr_cond_t filter;
   tr_state_t state;
   int copy_range = 0;
   int copy_current = 0;
   char option [1024];
   char range_s [1024];
   char expr [1024];

   if (isatty(0)) {
      printf ("error: expect stdin to be streaming data from ntracekd\n");
      exit(1);
   }

   if ( argc == 3 ) {
     sprintf(option,"%s",argv[1]);
     if (!strcmp(option,"--bracket")) {
       printf(detect_usage);
       exit (1);
     }

     sprintf(expr,"%s",argv[2]);
     sprintf(range_s,"%s",argv[1]);
     range = atoi(range_s);

     copy_current = 1;
   } else if ( argc == 4 ) {

     sprintf(option,"%s",argv[1]);
     if (strcmp(option,"--bracket")) {
       printf(detect_usage);
       exit (1);
     }

     sprintf(expr,"%s",argv[3]);
     sprintf(range_s,"%s",argv[2]);
     range = atoi(range_s);

     if (range <= 0) {
       printf("error: range must be greater than zero\n");
     }
     copy_range = 1;

   } else {
     printf(detect_usage);
     exit (1);
   }


   // Initialize the API
   t = tr_init();

   // Create a condition structure representing the users condition
   user = tr_cond_create(t,"user");
   tr_cond_expr_and(t,user,expr);
```

```
    // Create a state which starts when the condition true starts (which
    // will be true for the very first event and stops when the user's
    // condtion is met.
    start = tr_cond_create(t,"start");
    tr_cond_expr_and(t, start, "offset>=0");
    state = tr_state_create(t,"state");
    tr_state_start_cond(t,state,start);
    tr_state_end_cond(t,state,user);

    // Create a condition which is true when the state becomes inactive
    filter = tr_cond_create(t,"filter");
    tr_cond_expr_and(t, filter, "state_status(state)==0");

    // Open the input stream
    tr_open_stream(t, 0, 1024*1024*5, 0);

    if (copy_range){

      tr_cond_cb(t,filter,copy_input_range_cb,0);
      tr_iterate(t);

    } else if (copy_current){

      tr_cond_cb(t,filter,copy_current_input_cb,0);
      tr_iterate(t);

    }

    tr_close(t);

}

static
void
copy_input_range_cb
  (tr_t        t,
   tr_state_t  state,
   tr_offset_t offset,
   int         occurrence,
   void      * context,
   int       * disable)
{
  int i;
  int errs;
  tr_string_node_t * list;
  int start = offset - range;
  int end = offset + range;

  if (start <= 0) start = 0;
  if (end <= 0) end = 1;
  if (start == end) end++;

  tr_copy_input_range(t,"copy_input_range.data",0666,start,end);
  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));
  }
```

```
  *disable = 1;
}


static
void
copy_current_input_cb
  (tr_t        t,
   tr_state_t  state,
   tr_offset_t offset,
   int         occurrence,
   void      * context,
   int       * disable)
{
  int i;
  int errs;
  tr_string_node_t * list;
  int start = offset - range;
  int end = offset;

  if (start <= 0) start = 0;
  if (end <= 0) end = 1;
  if (start == end) end++;

  tr_copy_input_range(t,"copy_current_input.data",0666,start, end);
  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_halt(t);
}
```

F
Answers to Common Questions

What can I do if trace events are not logging at all?

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. Check the return codes from the API calls. See “trace_begin, Trace.begin” on page 2-7 for more information.

When should I log a different trace event ID number?

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, Trace.event and their variants” on page 2-13.

How can I prevent user trace events from being discarded or lost?

Use expansive mode; avoid use of buffer or file wrapping options. Flush the shared memory buffer more often by tuning:

- The shared memory buffer sizes
- The number of shared memory buffers
- Increase the priority of the user trace daemon
- Bind the user trace daemon to a CPU with minimal activity

See “Preventing Trace Event Loss” on page 6-1 and Chapter 3 for more information.

What can I do if trace events are not appearing in an ntrace display?

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 icon 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.

How can I prevent kernel trace events from being lost?

- Verify that the raw kernel trace output file (if not streaming) is on a local file system and not an NFS file system.
- Increase the size and number of the kernel trace buffers
- Increase the priority of the kernel trace daemon

- Bind the kernel trace daemon to a CPU with minimal activity

See "Preventing Trace Event Loss" on page 6-1 and Chapter 3 for more information.

Why can't I see my individual thread names?

In order to distinguish between threads, you must link with the thread-aware version of the NightTrace Logging API. You can name your threads with meaningful symbolic names by using the `trace_set_thread_name` routine. See "Threads and Logging" on page 2-33 for more information.

G
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.

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* or *profile*.

configuration file

An NightTrace-generated ASCII file that holds *display pages*, and *profile* definitions. This can also be a hand-edited table file, containing definition of *string tables* and/or *format tables*.

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)*.

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*. 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 *state* specified to the *function*, or the state being currently defined. Thus, if a qualfied 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 table

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and maps all known numeric *trace event ID*s 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, *function* calls, *and profile referneces.*

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-wrap-around 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.

lost trace event

A *trace event* **`ntraceud`** was unable to log. Several **`ntraceud`** options exist to prevent this trace event loss.

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 ocurrences 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, Ada, or Fortran; 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 NightTrace thread is either a process, an Ada task or a POSIX thread (or a set of any combination of these). The name of a thread is either a numeric value representing a threads internal identifier (usually its **`gettid(2)`** value), or a symbolic name assigned by various parties. For Ada tasks, the Ada runtime automatically names each thread (task) using its Ada-assigned name, when using the **`-trace`** or **`-ntrace`** link options. NightTrace defaults the name of the main thread of a process to "main". The user can set the name of a thread using `trace_set_thread_name`. See "Threads and Logging" on page 2-33 for more information.

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, Ada, or Fortran. 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, which is normally the value returned by getpid(2) for single-threaded applications, and gettid(2) for multi-threaded application in kernel data.

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.

profile

The "logical and" of several criteria such as event codes, processes, and threads. conditions used to identify an event or a state.

profile reference

The name of a *profile*.

pull-down menu

A set of critera defining conditions for an event or state; e.g event IDs, argument values, CPU, process, thread.

push button

A graphic image of a labeled button. *Click* on a push button to select it.

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.

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, profiles, 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 *state* specified to the *function*, or the state being currently defined. Thus, if a 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.

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*.

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 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 an internal NightTrace context to which *trace events* can be associated.

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.

time quantum

The fixed period of time for which the kernel allocates the CPU to a process.

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 *profile* 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*.

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 NightTrace 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*.

Index

Symbols

/usr/bin/ntracekd 4-1
/usr/bin/ntraceud 3-1
/usr/include/ntrace.h 2-1
/usr/lib/libntrace.a 2-34
/usr/lib/libntrace_thr.a 2-34
/usr/lib/NightTrace/illuminators 5-73
“wrapper” routines 5-2
‹!-- comment --› 5-101
‹config› 5-101
‹declare› 5-102
‹defaults› 5-102
‹exclude› 5-103
‹function› 5-103
‹group› 5-104
‹level› 5-105
‹options› 5-108
‹variable› 5-109
‹wrapper_file_scope› 5-110
‹wrapper_post› 5-110
‹wrapper_pre› 5-110
‹wrapper_real› 5-111
‹wrapper› 5-110

A

a.link 5-74
a.out 7-5
a.outAI 5-75
Ada language
    compiling and linking 2-34
Ada task identifier 16-8, 16-46, 16-88, 16-125, 16-166, 18-81
addr_args 5-106, 5-107
addr_ret 5-106, 5-107
aggregate_limit 5-106, 5-108
Application Illumination 5-1
`arg` function 16-4, 16-21
`arg_dbl` function 16-22, 16-23
`arg_long_dbl` function 16-24
`arg_long_long` function 16-25, 16-67
`arg1` function 7-22, 16-5, 16-190
`arg2` function 16-9
args 5-106
`avg` function 16-179

B

`blk_arg` function 16-26
`blk_arg_bits` function 16-27
`blk_arg_char` function 16-28
`blk_arg_dbl` function 16-29
`blk_arg_flt` function 16-30
`blk_arg_long` function 16-31
`blk_arg_long_bits` function 16-32
`blk_arg_long_dbl` function 16-33
`blk_arg_long_long` function 16-34
`blk_arg_long_ubits` function 16-35
`blk_arg_short` function 16-36
`blk_arg_string` function 16-37
`blk_arg_ubits` function 16-38
`blk_arg_uchar` function 16-39
`blk_arg_uint` function 16-40
`blk_arg_ulong_long` function 16-41
`blk_arg_ushort` function 16-42
boolean table 7-18
Box
    interrupt 17-14
    syscall 17-15
Box exception 17-14
BUFFER_LENGTH 5-53, 5-75
Buffer-wraparound mode 2-25

C

C language
    compiling and linking 2-34
    source considerations 2-1
call 5-1
caller 5-106
`CAP_IPC_LOCK` capability C-1
`CAP_SYS_NICE` capability C-1

Capabilities C-1
ccur_rt 5-3, 5-65
character entities 5-102
clock_synchronize(1M) command 2-13
Comments
    event-map file 7-11
config.xml 5-66
Configuration parameters
    Then-Expression 16-188
Conserving disk space 6-3
Constant string literals 7-22, 16-10, 16-186
Constant times 16-3
Context switch
    lines 17-13, 17-14, 17-15
Context-sensitive help 5-13, 8-28
`cpu` function 16-48
Current time line 17-12, 17-14, 17-15

D

daemons, streaming 9-15
Data Box 16-188, 17-14, 17-15, 17-16
Data Graph 12-12, 17-14
device table 7-19, 17-4, 17-18
device_nodename table 7-20, 17-19
Disabling
    library routines 2-20, 2-33
    trace events 2-22
    tracing 2-20, 2-33
Discarding trace events 2-25, F-1
Display object
    Data Box 16-188, 17-14, 17-15, 17-16
    Data Graph 17-14
    Event Graph 17-16
    State Graph 17-15, 17-16
Display object configuration parameters
    Then-Expression 16-188
Display page area
    interval scroll bar F-1
Duration
    `state` 16-135

E

Enabling
    trace events 2-22
End functions 16-97
`end_arg` function 16-100
`end_arg_dbl` function 16-101, 16-102
`end_arg_long_dbl` function 16-103
`end_arg_long_long` function 16-104
`end_blk_arg` function 16-105
`end_blk_arg_bits` function 16-106
`end_blk_arg_char` function 16-107
`end_blk_arg_dbl` function 16-108
`end_blk_arg_flt` function 16-109
`end_blk_arg_long` function 16-110
`end_blk_arg_long_bits` function 16-111
`end_blk_arg_long_dbl` function 16-112
`end_blk_arg_long_long` function 16-113
`end_blk_arg_long_ubits` function 16-114
`end_blk_arg_short` function 16-115
`end_blk_arg_string` function 16-116
`end_blk_arg_ubits` function 16-117
`end_blk_arg_uchar` function 16-118
`end_blk_arg_uint` function 16-119
`end_blk_arg_ulong_long` function 16-120
`end_blk_arg_ushort` function 16-121
`end_cpu` function 16-127
`end_id` function 16-99
`end_node_id` function 16-130
`end_node_name` function 16-133
`end_num_args` function 16-122
`end_offset` function 16-128
`end_pid` function 16-123
`end_pid_table_name` function 16-131
`end_task_id` function 16-125
`end_thread_id` function 16-124
`end_tid` function 16-126
`end_tid_table_name` function 16-132
`end_time` function 16-129
Environment variable
    `NSLM_SERVER` A-2
errno 5-106, 5-107, 18-138, 18-139
Event
    gap 16-58
    matches 16-59
    qualified 16-193
Event Graph 12-5, 12-10, 17-16
Event ID. see Trace event ID
event table 7-17
Event. see Trace event
`event_gap` function 16-58
event_ids 5-108
`event_matches` function 16-59
Event-map file 2-16, 7-2, 7-11
Exception 17-3, 17-14, 17-17, 17-19
    graph 17-14
    resumption 17-14
    suspension 17-14
Exception box 17-14
exclude 5-106, 5-107
execve(2) service 2-11

Expressions
constant string literals 7-22, 16-10, 16-186
functions 16-4
operands 16-1
operators 16-1

F

File
/usr/bin/ntracekd 4-1
/usr/bin/ntraceud 3-1
/usr/include/ntrace.h 2-1
/usr/lib/libntrace.a 2-34
/usr/lib/libntrace_thr.a 2-34
event-map 2-16, 7-2, 7-11
trace event 2-8, 3-1, 7-10
vectors 7-17, 17-2, 17-17, 17-18, 17-19
File system
NFS F-1
filename 5-109
Fixed licenses A-1
Floating licenses A-1
Flushing shared memory buffer 2-24
fork(2) service 2-11
Format
functions 16-184
format function 16-190
Format table 7-20, 16-188
get_format function 16-188
Fortran language
compiling and linking 2-34, 2-35
frame 5-106
function call 5-1
function return 5-1
Functions 16-4
arg 16-4, 16-21
arg_dbl 16-22, 16-23
arg_long_dbl 16-24
arg_long_long 16-25, 16-67
arg1 7-22, 16-5, 16-190
arg2 16-9
avg 16-179
blk_arg 16-26
blk_arg_bits 16-27
blk_arg_char 16-28
blk_arg_dbl 16-29
blk_arg_flt 16-30
blk_arg_long 16-31
blk_arg_long_bits 16-32
blk_arg_long_dbl 16-33
blk_arg_long_long 16-34
blk_arg_long_ubits 16-35
blk_arg_short 16-36
blk_arg_string 16-37
blk_arg_ubits 16-38
blk_arg_uchar 16-39
blk_arg_uint 16-40
blk_arg_ulong_long 16-41
blk_arg_ushort 16-42
cpu 16-48
end 16-97
end_arg 16-100
end_arg_dbl 16-101, 16-102
end_arg_long_dbl 16-103
end_arg_long_long 16-104
end_blk_arg 16-105
end_blk_arg_bits 16-106
end_blk_arg_char 16-107
end_blk_arg_dbl 16-108
end_blk_arg_flt 16-109
end_blk_arg_long 16-110
end_blk_arg_long_bits 16-111
end_blk_arg_long_dbl 16-112
end_blk_arg_long_long 16-113
end_blk_arg_long_ubits 16-114
end_blk_arg_short 16-115
end_blk_arg_string 16-116
end_blk_arg_ubits 16-117
end_blk_arg_uchar 16-118
end_blk_arg_uint 16-119
end_blk_arg_ulong_long 16-120
end_blk_arg_ushort 16-121
end_cpu 16-127
end_id 16-99
end_node_id 16-130
end_node_name 16-133
end_num_args 16-122
end_offset 16-128
end_pid 16-123
end_pid_table_name 16-131
end_task_id 16-125
end_thread_id 16-124
end_tid 16-126
end_tid_table_name 16-132
end_time 16-129
event_gap 16-58
event_matches 16-59
format 16-184
format 16-190
get_format 16-188
get_item 16-186
get_string 7-22, 16-184
id 16-20, 16-188, 16-190
lookup_pc 16-191
max 16-178
max_offset 16-182

min 16-177
min_offset 16-181
multi-event 16-58
multi-state 16-134
node_id 16-51
node_name 16-54
num_args 16-43
offset 16-138
offset 7-22, 16-49
offset_arg 16-141
offset_arg_dbl 16-142, 16-143
offset_arg_long_dbl 16-144
offset_arg_long_long 16-145
offset_blk_arg 16-146
offset_blk_arg_bits 16-147
offset_blk_arg_char 16-148
offset_blk_arg_dbl 16-149
offset_blk_arg_flt 16-150
offset_blk_arg_long 16-151
offset_blk_arg_long_bits 16-152
offset_blk_arg_long_dbl 16-153
offset_blk_arg_long_long 16-154
offset_blk_arg_long_ubits 16-155
offset_blk_arg_short 16-156
offset_blk_arg_string 16-157
offset_blk_arg_ubits 16-158
offset_blk_arg_uchar 16-159
offset_blk_arg_uint 16-160
offset_blk_arg_ulong_long 16-161
offset_blk_arg_ushort 16-162
offset_cpu 16-168
offset_id 16-140, 16-181, 16-182
offset_node_id 16-170
offset_node_name 16-173
offset_num_args 16-163
offset_pid 16-164
offset_pid_table_name 16-171
offset_process_name 16-174
offset_task_id 16-166
offset_task_name 16-175
offset_thread_id 16-165
offset_thread_name 16-176
offset_tid 16-167
offset_tid_table_name 16-172
offset_time 16-169
pid 16-44, 16-188
pid_table_name 16-52
process_name 16-55
start 16-60
start_arg 16-63
start_arg_dbl 16-64, 16-65
start_arg_long_dbl 16-66
start_blk_arg 16-68
start_blk_arg_bits 16-69
start_blk_arg_char 16-70
start_blk_arg_dbl 16-71
start_blk_arg_flt 16-72
start_blk_arg_long 16-73
start_blk_arg_long_bits 16-74
start_blk_arg_long_dbl 16-75
start_blk_arg_long_long 16-76
start_blk_arg_long_ubits 16-77
start_blk_arg_short 16-78
start_blk_arg_string 16-79
start_blk_arg_ubits 16-80
start_blk_arg_uchar 16-81
start_blk_arg_uint 16-82
start_blk_arg_ulong_long 16-83
start_blk_arg_ushort 16-84
start_cpu 16-90
start_id 16-5, 16-62
start_node_id 16-93
start_node_name 16-96
start_num_args 16-85
start_offset 16-91
start_pid 16-86
start_pid_table_name 16-94
start_task_id 16-88
start_thread_id 16-87
start_tid 16-89
start_tid_table_name 16-95
start_time 16-92
state_dur 16-135
state_gap 16-5, 16-134
state_matches 16-136
state_status 16-137
string 16-16
sum 16-180
summary 16-177
summary_matches 16-183
table 16-184
task_id 16-46
task_name 16-56
thread_id 16-45
thread_name 16-57
tid 16-47
tid_table_name 16-53
time 16-50
trace event 16-18

G

Gap
event 16-58
state 16-134
get_format function 16-188

`get_item` function 16-186
`get_string` function 7-22, 16-184
glibc 5-3, 5-64
Global process identifier 16-7, 16-44
Graph
    data 12-12, 17-14
    event 12-5, 12-10, 17-16
    exception 17-14
    interrupt 17-14
    state 12-11, 17-15, 17-16
    syscall 17-15

H

Hardclock interrupts 17-14
Help
    On Context 5-13, 8-28

I

`id` function 16-20, 16-188, 16-190
illuminator
    definition 5-2
    entry event 5-2
    return event 5-2
illuminator.h 5-66
illuminator.map 5-66
illuminator.o 5-67
illuminator.vararg 5-67
illuminator_level.fmt 5-66
illuminator_level.list 5-67
illuminator_level.o 5-66
Inter-process communication 2-6
Interrupt 17-2, 17-14, 17-17, 17-19
    graph 17-14
    hardclock 17-14
Interrupt box 17-14
Interval
    scroll bar F-1
iregex 5-109
IRQ_ENTRY trace event 17-2
IRQ_EXIT trace event 17-3

J

Java 2-3

K

Kernel tracing 7-17, 7-18, 17-1

L

Language
    Ada 2-34
    C 2-1, 2-34
    Fortran 2-34, 2-35
    Java 2-3
libntrace.a 2-34
libntrace_tjr.a 2-34
Library routines 2-1
    overloading in Ada 2-3
    return values 2-2
    `Trace.begin` 2-7, 2-18, 2-23, 2-30
    `Trace.closeThread` 2-27
    `Trace.disable` 2-20
    `Trace.enable` 2-20
    `Trace.end` 2-12, 2-29
    `Trace.flush` 2-24
    `Trace.setThreadName` 2-26
    `Trace.trigger` 2-24
    `trace_begin` 2-7, 2-18, 2-23, 2-30, 3-1, F-1
    `trace_close_thread` 2-27
    `trace_default_config` 2-8
    `trace_disable` 2-20
    `trace_disable_all` 2-20, 2-33
    `trace_disable_range` 2-20
    `trace_enable` 2-20
    `trace_enable_all` 2-20
    `trace_enable_range` 2-20
    `trace_end` 2-12, 2-29, 3-2
    `trace_event` 2-13
    `trace_flush` 2-24, 3-2
    `trace_set_thread_name` 2-26
    `trace_trigger` 2-24, 3-2
licences 1-1
License A-1
    fixed A-1
    installation A-1
    keys A-1
    modes A-1
    `nslm_admin` A-1, A-3
    report A-3
    requests A-2
    server A-2
    support A-4
License manager 1-1
-lntrace 5-73

-lntrace_thr 5-73
Loading
    trace event 7-6
Logging
    trace event 6-4, F-1
`lookup_pc` function 16-191
Loss
    trace event 2-18, F-1

M

Macros 16-193
main 5-3
Map file. see Event-map file
Matches
    event 16-59
    state 16-136
    summary 16-183
`max` function 16-178
`max_offset` function 16-182
Maximum value 16-178, 16-182
Menu option
    On Context 5-13, 8-28
    On Help 8-28, 8-29
`min` function 16-177
`min_offset` function 16-181
Minimum value 16-177, 16-181
Mode
    buffer-wraparound 2-25
Multi-event functions 16-58
Multi-state functions 16-134

N

name_pid table 7-18, 17-18
name_tid table 7-18
next_event.txt 5-66
NFS file system F-1
NightLight, see nlight
NightStar Licence Manager 1-1
NightTrace thread identifier 16-8, 16-47, 16-89, 16-126, 16-167, 18-78
nlight 5-2
    --ada 5-74
    --aggregate_limit 5-68
    --build 5-72
    --cf77 5-74
    commands 5-68
    --config 5-69
    --create 5-68
    detail levels 5-3
    --do_nodebug 5-69
    --dont_nodebug 5-69
    --event_ids 5-69
    --g77 5-74
    --gcc 5-73
    GUI 5-5
    --i 5-70
    --illuminate 5-74
    --install 5-69
    --iregex 5-70
    --istd 5-71
    --iunderscores 5-70
    main illuminator 5-53
    --populate 5-71
    --report 5-72
    session manager 5-37
    wizard 5-15
    work flow 5-37
    work flow illustration 5-2
    --x 5-70
    --xregex 5-70
    --xstd 5-71
    --xunderscores 5-70
NLSM 1-1
Node identifer 16-51
Node identifier
    ending trace event 16-130
    offset 16-170
    starting trace event 16-93
Node name 16-54
    ending trace event 16-133
    ordinal trace event 16-173
    starting trace event 16-96
`node_id` function 16-51
`node_name` function 16-54
node_name table 7-19, 17-18
nodebug 5-108
`nslm_admin` A-1, A-3
`NSLM_SERVER` A-2
ntrace 1-4
    format tables 7-20
    functions 16-4
    operands 16-1
    operators 16-1
    performance considerations 7-6
    string tables 7-15
ntrace functions 16-4
ntrace option
    --end (load events before constraint) 7-4
    --listing (list trace events) 7-12
    --start (load events after constraint) 7-4
ntrace qualified states 16-62, 16-63, 16-64, 16-65, 16-66, 16-67, 16-68, 16-69, 16-70, 16-71,

16-72, 16-73, 16-74, 16-75, 16-76, 16-77, 16-78, 16-79, 16-80, 16-81, 16-82, 16-83, 16-84, 16-85, 16-86, 16-87, 16-88, 16-89, 16-90, 16-91, 16-92, 16-93, 16-94, 16-95, 16-96, 16-97, 16-99, 16-100, 16-101, 16-102, 16-103, 16-104, 16-105, 16-106, 16-107, 16-108, 16-109, 16-110, 16-111, 16-112, 16-113, 16-114, 16-115, 16-116, 16-117, 16-118, 16-119, 16-120, 16-121, 16-122, 16-123, 16-124, 16-125, 16-126, 16-127, 16-128, 16-129, 16-130, 16-131, 16-132, 16-133, 16-134, 16-135, 16-136, 16-137
ntrace.h 2-1
ntracekd
daemon 4-1
ntraceud
daemon 3-1
invoking 3-6
ntraceud mode
buffer-wraparound 2-25
num_args function 16-43
NUM_BUFFERS 5-53, 5-75

O

Offset 7-4, 16-4, 16-9, 16-10, 16-138, 16-140, 16-141, 16-142, 16-143, 16-144, 16-145, 16-146, 16-147, 16-148, 16-149, 16-150, 16-151, 16-152, 16-153, 16-154, 16-155, 16-156, 16-157, 16-158, 16-159, 16-160, 16-161, 16-162, 16-163, 16-164, 16-165, 16-166, 16-167, 16-168, 16-169, 16-170, 16-171, 16-172, 16-173, 16-174, 16-175, 16-176
offset function 7-22, 16-49
Offset functions 16-138
offset_arg function 16-141
offset_arg_dbl function 16-142, 16-143
offset_arg_long_dbl function 16-144
offset_arg_long_long function 16-145
offset_blk_arg function 16-146
offset_blk_arg_bits function 16-147
offset_blk_arg_char function 16-148
offset_blk_arg_dbl function 16-149
offset_blk_arg_flt function 16-150
offset_blk_arg_long function 16-151
offset_blk_arg_long_bits function 16-152
offset_blk_arg_long_dbl function 16-153
offset_blk_arg_long_long function 16-154
offset_blk_arg_long_ubits function 16-155
offset_blk_arg_short function 16-156
offset_blk_arg_string function 16-157
offset_blk_arg_ubits function 16-158
offset_blk_arg_uchar function 16-159
offset_blk_arg_uint function 16-160
offset_blk_arg_ulong_long function 16-161
offset_blk_arg_ushort function 16-162
offset_cpu function 16-168
offset_id function 16-140, 16-181, 16-182
offset_node_id function 16-170
offset_node_name function 16-173
offset_num_args function 16-163
offset_pid function 16-164
offset_pid_table_name function 16-171
offset_process_name function 16-174
offset_task_id function 16-166
offset_task_name function 16-175
offset_thread_id function 16-165
offset_thread_name function 16-176
offset_tid function 16-167
offset_tid_table_name function 16-172
offset_time function 16-169
On Context menu option 5-13, 8-28
On Help menu option 8-28, 8-29
Operands
constants 16-2
functions 16-4
qualified states 16-62, 16-63, 16-64, 16-65, 16-66, 16-67, 16-68, 16-69, 16-70, 16-71, 16-72, 16-73, 16-74, 16-75, 16-76, 16-77, 16-78, 16-79, 16-80, 16-81, 16-82, 16-83, 16-84, 16-85, 16-86, 16-87, 16-88, 16-89, 16-90, 16-91, 16-92, 16-93, 16-94, 16-95, 16-96, 16-97, 16-99, 16-100, 16-101, 16-102, 16-103, 16-104, 16-105, 16-106, 16-107, 16-108, 16-109, 16-110, 16-111, 16-112, 16-113, 16-114, 16-115, 16-116, 16-117, 16-118, 16-119, 16-120, 16-121, 16-122, 16-123, 16-124, 16-125, 16-126, 16-127, 16-128, 16-129, 16-130, 16-131, 16-132, 16-133, 16-134, 16-135, 16-136, 16-137
Operands in expressions 16-1
Operators in expressions 16-1

P

Performance considerations
ntrace 7-6
PID 16-7, 16-44
pid function 16-44, 16-188
pid table 7-17, 17-19
PID table name 16-52
pid_nodename table 7-19, 17-18
pid_table_name function 16-52
Pre-defined tables 7-17, 17-4, 17-17

printf(3) routine 7-13, 7-21
printf(3S) routine 16-190
Privileged access C-1
Process identifier
    ending trace event 16-131
    offset 16-171
    starting trace event 16-94
Process identifier table name 16-52
Process name 16-55
    ordinal trace event 16-174
`process_name` function 16-55
pthread 5-3, 5-65
Push button
    Zoom Out F-1

Q

Qualified events 16-193
Qualified states 16-62, 16-63, 16-64, 16-65, 16-66, 16-67, 16-68, 16-69, 16-70, 16-71, 16-72, 16-73, 16-74, 16-75, 16-76, 16-77, 16-78, 16-79, 16-80, 16-81, 16-82, 16-83, 16-84, 16-85, 16-86, 16-87, 16-88, 16-89, 16-90, 16-91, 16-92, 16-93, 16-94, 16-95, 16-96, 16-97, 16-99, 16-100, 16-101, 16-102, 16-103, 16-104, 16-105, 16-106, 16-107, 16-108, 16-109, 16-110, 16-111, 16-112, 16-113, 16-114, 16-115, 16-116, 16-117, 16-118, 16-119, 16-120, 16-121, 16-122, 16-123, 16-124, 16-125, 16-126, 16-127, 16-128, 16-129, 16-130, 16-131, 16-132, 16-133, 16-134, 16-135, 16-136, 16-137

R

Record. see Trace event
return 5-1
Return values 2-2
return_val 5-106, 5-107

S

SCHED_CHANGE trace event 17-2
Scroll bar F-1
Shared memory
    failure to attach 2-13
    flushing 2-24
SOFT_IRQ_ENTRY trace event 17-3
SOFT_IRQ_EXIT trace event 17-3
Start functions 16-60
`start_arg` function 16-63
`start_arg_dbl` function 16-64, 16-65
`start_arg_long_dbl` function 16-66
`start_blk_arg` function 16-68
`start_blk_arg_bits` function 16-69
`start_blk_arg_char` function 16-70
`start_blk_arg_dbl` function 16-71
`start_blk_arg_flt` function 16-72
`start_blk_arg_long` function 16-73
`start_blk_arg_long_bits` function 16-74
`start_blk_arg_long_dbl` function 16-75
`start_blk_arg_long_long` function 16-76
`start_blk_arg_long_ubits` function 16-77
`start_blk_arg_short` function 16-78
`start_blk_arg_string` function 16-79
`start_blk_arg_ubits` function 16-80
`start_blk_arg_uchar` function 16-81
`start_blk_arg_uint` function 16-82
`start_blk_arg_ulong_long` function 16-83
`start_blk_arg_ushort` function 16-84
`start_cpu` function 16-90
`start_id` function 16-5, 16-62
`start_node_id` function 16-93
`start_node_name` function 16-96
`start_num_args` function 16-85
`start_offset` function 16-91
`start_pid` function 16-86
`start_pid_table_name` function 16-94
`start_task_id` function 16-88
`start_thread_id` function 16-87
`start_tid` function 16-89
`start_tid_table_name` function 16-95
`start_time` function 16-92
State 2-17, 17-14
    duration 16-135
    gap 16-134
    matches 16-136
State Graph 12-11, 17-15, 17-16
`state_dur` function 16-135
`state_gap` function 16-5, 16-134
`state_matches` function 16-136
`state_status` function 16-137
Statistics
    multi-event 16-58
    multi-state 16-134
    summary 16-177
std 5-109
`strcmp function` 16-16
streaming daemons 9-15
String functions
    `strcmp` 16-16
    `strncmp` 16-17

String table 7-15, 16-184, 16-186
boolean 7-18
device 7-19, 17-4, 17-18
device_nodename 7-20, 17-19
event 7-17
`get_item` function 16-186
`get_string` function 7-22, 16-184
name_pid 7-18, 17-18
name_tid 7-18
node_name 7-19, 17-18
pid 7-17, 17-19
pid_nodename 7-19, 17-18
syscall 7-19, 17-4, 17-17
syscall_nodename 7-19, 17-18
tid 7-18
tid_nodename 7-19
vector 7-19, 17-2, 17-3, 17-17
vector_nodename 7-19, 17-19
`strncmp function` 16-17
substitution variables 5-48
`sum` function 16-180
Summary
matches 16-183
Summary functions 16-177
`summary_matches` function 16-183
Syscall 17-4, 17-15, 17-17
graph 17-15
suspension 17-15
Syscall box 17-15
syscall table 7-19, 17-4, 17-17
SYSCALL_EXIT trace event 17-4
syscall_nodename table 7-19, 17-18
SYSCALL_RESUME trace event 17-5
SYSCALL_SUSPEND trace event 17-5
System call 17-4, 17-15, 17-17

T

Table
boolean 7-18
device 7-19, 17-4, 17-18
device_nodename 7-20, 17-19
event 7-17
format 7-20, 16-188
functions 16-184
name_pid 7-18, 17-18
name_tid 7-18
node_name 7-19, 17-18
pid 7-17, 17-19
pid_nodename 7-19, 17-18
pre-defined 7-17, 17-4, 17-17
string 7-15, 16-184, 16-186
syscall 7-19, 17-4, 17-17
syscall_nodename 7-19, 17-18
tid 7-18
tid_nodename 7-19
vector 7-19, 17-2, 17-3, 17-17
vector_nodename 7-19, 17-19
Task name 16-56
ordinal trace event 16-175
`task_id` function 16-46
`task_name` function 16-56
Then-Expression configuration parameter 16-188
Thread event
ordinal 16-172
Thread identifier
ending trace event 16-132
offset 16-172
starting trace event 16-95
Thread identifier table name 16-53
Thread name 16-57
ordinal trace event 16-176
Thread names 7-2, 7-18
`thread_id` function 16-45
`thread_name` function 16-57
TID 16-8, 16-47, 16-89, 16-126, 16-167, 18-78
`tid` function 16-47
tid table 7-18
TID table name 16-53
tid_nodename table 7-19
`tid_table_name` function 16-53
`time` function 16-50
timeline panels 12-1
Times
constant 16-3
Timestamp 7-2, 16-50, 16-92, 16-129, 16-169
tr_activate() 18-133
tr_append_table() 18-143
tr_arg_dbl() 18-38, 18-45
tr_arg_dbl_() 18-38, 18-45
tr_arg_int() 18-36
tr_arg_int_() 18-37, 18-44
tr_arg_long() 18-39, 18-46
tr_arg_long_() 18-40, 18-47
tr_arg_long_dbl() 18-41, 18-48
tr_arg_long_dbl_() 18-41, 18-48
tr_arg_long_long() 18-42, 18-49
tr_arg_long_long_() 18-43, 18-50
tr_arg_t 18-2
tr_argtype 18-50
tr_argtype_ 18-51
tr_blk_arg() 18-51
tr_blk_arg_() 18-52
tr_blk_arg_bits() 18-53
tr_blk_arg_bits_() 18-54
tr_blk_arg_char() 18-55, 18-73

tr_blk_arg_char_() 18-55
tr_blk_arg_dbl() 18-56
tr_blk_arg_dbl_() 18-57
tr_blk_arg_flt() 18-58
tr_blk_arg_flt_() 18-58
tr_blk_arg_long() 18-59
tr_blk_arg_long_() 18-60
tr_blk_arg_long_bits) 18-61
tr_blk_arg_long_bits_() 18-62
tr_blk_arg_long_dbl() 18-63
tr_blk_arg_long_dbl_() 18-63
tr_blk_arg_long_long() 18-64
tr_blk_arg_long_long_() 18-65
tr_blk_arg_long_ubits() 18-66
tr_blk_arg_long_ubits_() 18-67
tr_blk_arg_short() 18-68
tr_blk_arg_string() 18-69
tr_blk_arg_string_() 18-70
tr_blk_arg_ubits() 18-71
tr_blk_arg_ubits_() 18-72
tr_blk_arg_uchar_() 18-74
tr_blk_arg_ushort() 18-75
tr_blk_arg_ushort_() 18-68, 18-75
tr_cancel_cb() 18-146
tr_cb_t 18-3
tr_close() 18-20
tr_cond_and() 18-115
tr_cond_cb() 18-147
tr_cond_cb_func_t 18-3
tr_cond_copy() 18-116
tr_cond_cpu() 18-96
tr_cond_cpu_clear() 18-97
tr_cond_create() 18-91
tr_cond_expr_and() 18-111
tr_cond_expr_or() 18-112
tr_cond_find() 18-92
tr_cond_func_and() 18-108
tr_cond_func_clear() 18-110
tr_cond_func_or() 18-106
tr_cond_func_t 18-4
tr_cond_id() 18-93
tr_cond_id_clear() 18-95
tr_cond_id_range() 18-94
tr_cond_name() 18-118
tr_cond_node() 18-104
tr_cond_node_clear() 18-105
tr_cond_not() 18-113
tr_cond_offset() 18-121
tr_cond_or() 18-114
tr_cond_pid() 18-98
tr_cond_pid_clear() 18-100
tr_cond_pid_name() 18-99
tr_cond_register() 18-120
tr_cond_reset() 18-92
tr_cond_satisfy() 18-118
tr_cond_satisfy_() 18-119
tr_cond_t 18-4
tr_cond_tid() 18-101
tr_cond_tid_clear() 18-103
tr_cond_tid_name() 18-102
tr_copy_input() 18-138
tr_copy_input_range() 18-139
tr_cpu() 18-82
tr_cpu_() 18-83
tr_create_table() 18-142
tr_destroy() 18-14
tr_dir_t 18-4
TR_EOF 18-4, 18-25, 18-26, 18-27, 18-28, 18-121, 18-134, 18-135
tr_error_check() 18-17
tr_error_clear() 18-16
tr_free() 18-24
tr_get_item() 18-141
tr_get_string() 18-140
tr_halt() 18-146
tr_id() 18-32
tr_id_() 18-32
tr_init() 18-14
tr_iterate() 18-145
tr_nargs() 18-35
tr_nargs_() 18-35
tr_next_event() 18-25
tr_next_event_() 18-26
TR_NO_CB 18-147, 18-148
TR_NO_COND 18-91, 18-93, 18-113, 18-115, 18-116, 18-117
TR_NO_HANDLE 18-14
TR_NO_STATE 18-123, 18-124
tr_node() 18-84
tr_node_() 18-84
tr_offset_t 18-4
tr_open_file() 18-18
tr_open_stream() 18-19
tr_pid() 18-76
tr_pid_() 18-77
tr_prev_event() 18-26
tr_prev_event_() 18-27
tr_process_name() 18-85
tr_process_name_() 18-86
tr_search() 18-28
tr_seek() 18-29
tr_state_action_t 18-5
tr_state_active() 18-136
tr_state_active_() 18-137
tr_state_cb() 18-148
tr_state_cb_func_t 18-5
tr_state_create() 18-122
tr_state_end_cond() 18-131

tr_state_end_cond_clear() 18-132
tr_state_end_id() 18-127
tr_state_end_id_clear() 18-129
tr_state_end_id_range() 18-128
tr_state_find() 18-123
tr_state_info() 18-134
tr_state_info_() 18-135
tr_state_info_t 18-6
tr_state_name() 18-124
tr_state_start_cond() 18-130
tr_state_start_cond_clear() 18-131
tr_state_start_id() 18-125
tr_state_start_id_clear() 18-127
tr_state_start_id_range() 18-126
tr_state_t 18-7
tr_stream_event_t 18-7
tr_stream_func_t 18-7
tr_stream_notify() 18-21
tr_stream_read() 18-22
TR_STREAM_SAVE 18-19
tr_stream_size() 18-23
tr_string_node 18-7
TR_SYSCALL_ENTRY trace event 17-4
tr_t 18-8
tr_task_id() 18-81
tr_task_id_() 18-81
tr_task_name() 18-86
tr_task_name_() 18-87
tr_thread_id() 18-79
tr_thread_id_() 18-80
tr_thread_name() 18-88
tr_thread_name_() 18-88
tr_tid() 18-78
tr_tid_() 18-78
tr_time() 18-33
tr_time_() 18-34
Trace event 1-2
  arguments 2-16, 7-2, 7-12, 7-14, 16-21, 16-22, 16-23, 16-24, 16-25, 16-26, 16-27, 16-28, 16-29, 16-30, 16-31, 16-32, 16-33, 16-34, 16-35, 16-36, 16-37, 16-38, 16-39, 16-40, 16-41, 16-42, 16-43, 16-63, 16-64, 16-65, 16-66, 16-67, 16-68, 16-69, 16-70, 16-71, 16-72, 16-73, 16-74, 16-75, 16-76, 16-77, 16-78, 16-79, 16-80, 16-81, 16-82, 16-83, 16-84, 16-85, 16-100, 16-101, 16-102, 16-103, 16-104, 16-105, 16-106, 16-107, 16-108, 16-109, 16-110, 16-111, 16-112, 16-113, 16-114, 16-115, 16-116, 16-117, 16-118, 16-119, 16-120, 16-121, 16-122, 16-141, 16-142, 16-143, 16-144, 16-145, 16-146, 16-147, 16-148, 16-149, 16-150, 16-151, 16-152, 16-153, 16-154, 16-155, 16-156, 16-157, 16-158, 16-159, 16-160, 16-161, 16-162, 16-163
  context switch 17-2
  disabling 2-22
  discarding 2-25, F-1
  enabling 2-22
  exception 17-3
  file 2-8, 3-1, 7-10
  functions 16-18
  ID 1-2, 2-16, 2-21, 7-2, 7-10, 7-12, F-1
  information 16-18
  interrupt 17-2
  IRQ_ENTRY 17-2
  IRQ_EXIT 17-3
  loading 7-6
  logging 6-4, F-1
  loss 2-18, F-1
  node identifer (ending trace event) 16-130
  node identifer (offset) 16-170
  node identifer (starting trace event) 16-93
  node identifier 16-51
  node name 16-54
  node name (ending trace event) 16-133
  node name (ordinal trace event) 16-173
  node name (starting trace event) 16-96
  offset 16-138
  offset. see Offset
  ordinal 16-170, 16-171, 16-173, 16-174, 16-175, 16-176
  ordinal number. see Offset
  PID table name 16-52
  process identifer (ending trace event) 16-131
  process identifer (offset) 16-171
  process identifer (starting trace event) 16-94
  process identifier table name 16-52
  process name 16-55
  process name (ordinal trace event) 16-174
  SCHED_CHANGE 17-2
  SOFT_IRQ_ENTRY 17-3
  SOFT_IRQ_EXIT 17-3
  syscall 17-4
  SYSCALL_EXIT 17-4
  SYSCALL_RESUME 17-5
  SYSCALL_SUSPEND 17-5
  task name 16-56
  task name (ordinal trace event) 16-175
  thread identifer (ending trace event) 16-132
  thread identifer (offset) 16-172
  thread identifer (starting trace event) 16-95
  thread identifier table name 16-53
  thread name 16-57
  thread name (ordinal trace event) 16-176
  TID table name 16-53
  timestamp 7-2, 16-50, 16-92, 16-129, 16-169
  TR_SYSCALL_ENTRY 17-4

TRAP_ENTRY 17-3
TRAP_EXIT 17-4
TRAP_RESUME 17-4
TRAP_SUSPEND 17-4
Trace point 1-2, 2-16
`Trace.begin` 2-7, 2-18, 2-23, 2-30
`Trace.closeThread` 2-27
`Trace.disable` 2-20
`Trace.enable` 2-20
`Trace.end` 2-12, 2-29
`Trace.flush` 2-24
`Trace.setThreadName` 2-26
`Trace.trigger` 2-24
`trace_begin` 2-7, 2-18, 2-23, 2-30, 3-1, F-1
`trace_close_thread` 2-27
`trace_default_config` 2-8
`trace_disable` 2-20
`trace_disable_all` 2-20, 2-33
`trace_disable_range` 2-20
`trace_enable` 2-20
`trace_enable_all` 2-20
`trace_enable_range` 2-20
`trace_end` 2-12, 2-29, 3-2
`trace_event` 2-13
TRACE_FILE 5-53, 5-75
`trace_flush` 2-24, 3-2
`trace_set_thread_name` 2-26
`trace_trigger` 2-24, 3-2
Tracing
disabling 2-20, 2-33
kernel 7-17, 7-18, 17-1
TRAP_ENTRY trace event 17-3
TRAP_EXIT trace event 17-4
TRAP_RESUME trace event 17-4
TRAP_SUSPEND trace event 17-4
triggers 9-15

U

underscores 5-108

V

variables 5-106, 5-107
vector table 7-19, 17-2, 17-3, 17-17
vector_nodename table 7-19, 17-19
vectors file 7-17, 17-2, 17-17, 17-18, 17-19

W

-Wl,--emit-relocs 5-73

X

xregex 5-109

Z

Zoom Out push button F-1