NightTrace RT User's Guide

Version 6.1

(RedHawk™ Linux®)

concurrent

0898398-150
July 2006

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

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

Preface

Scope of Manual

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

Structure of Manual

The manual includes for major parts as shown below:

- Part I - Event Logging and Capture
- Part II - Graphical Analysis
- Part III - Programmatic Analysis
- Part IV - Reference

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.
emphasis	Words or phrases that require extra emphasis use emphasis type.

window	Keyboard sequences and window features such as button, field, and menu labels and window titles appear in window type.
[]	Brackets enclose command options and arguments that are optional. You do not type the brackets if you choose to specify such option or arguments.
{}	Braces enclose mutually exclusive choices separated by the pipe (|) character, where one choice must be selected. You do not type the braces or the pipe character with the choice.
. . .	An ellipsis follows an item that can be repeated.

Referenced Publications

The following publications are referenced in this document:

0890240	*Concurrent Fortran 77 Reference Manual*
0890300	*X Window® System User's Guide*
0890378	*C: A Reference Manual*
0890380	*OSF/Motif™ Documentation Set* (3 volumes)
0898008	*NightStar RT Installation Guide*
0898008	*NightStar RT Tutorial*
0898398	*NightTrace RT User's Guide*
0898465	*NightProbe RT User's Guide*
0898480	*NightSim RT User's Guide*
0898515	*NightTune RT User's Guide*
0898537	*MAXAda™ for Linux Reference Manual*
0891019	*Concurrent C Reference Manual*
0891082	*Real-Time Clock and Interrupt Module User's Guide*

Contents

Chapter 4 Capturing Kernel Events with ntracekd

Chapter 5 Performance Tuning

Part II - Graphical Analysis

Chapter 6 Invoking NightTrace

Chapter 7 The NightTrace Main Window

Chapter 8 Profiles

Chapter 9 Display Pages

Chapter 10 Display Objects

Chapter 11 Using Expressions

Chapter 12 Kernel Tracing

Part III - Programmatic Analysis

Chapter 13 Using the NightTrace Analysis API

Appendix B Kernel Dependencies

Appendix C NightTrace Logging API Examples

Appendix D NightTrace Analysis API Examples

Appendix E Answers to Common Questions

Appendix F Glossary

Illustrations

Tables

Index

1
Introduction

NightTrace is a member of the NightStar™ RT 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

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

NightTrace operates with the all flavors of the RedHawk kernel; standard, tracing, and debug. In order to use kernel tracing, you must select the tracing or debug kernel at boot time from the boot-loader menu.

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 infor-

mation 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

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

The RedHawk kernel is built with kernel 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. However, the Real-Time Clock and Interrupt Module (RCIM) can be also used as a timestamp source.

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

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

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 and for individual events or user-defined states. Summaries can be generated via command line invocation of **`ntrace`** for generating automated reports.

Part I - Event Logging and Capture

Part I Event Logging and Capture

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).

Language-Specific Source Considerations

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

The NightTrace Logging API can be used with the following compilers:

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

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

C

NightTrace applications written in C 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 2-25)

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 2-26 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.*

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 Figure 2-1.

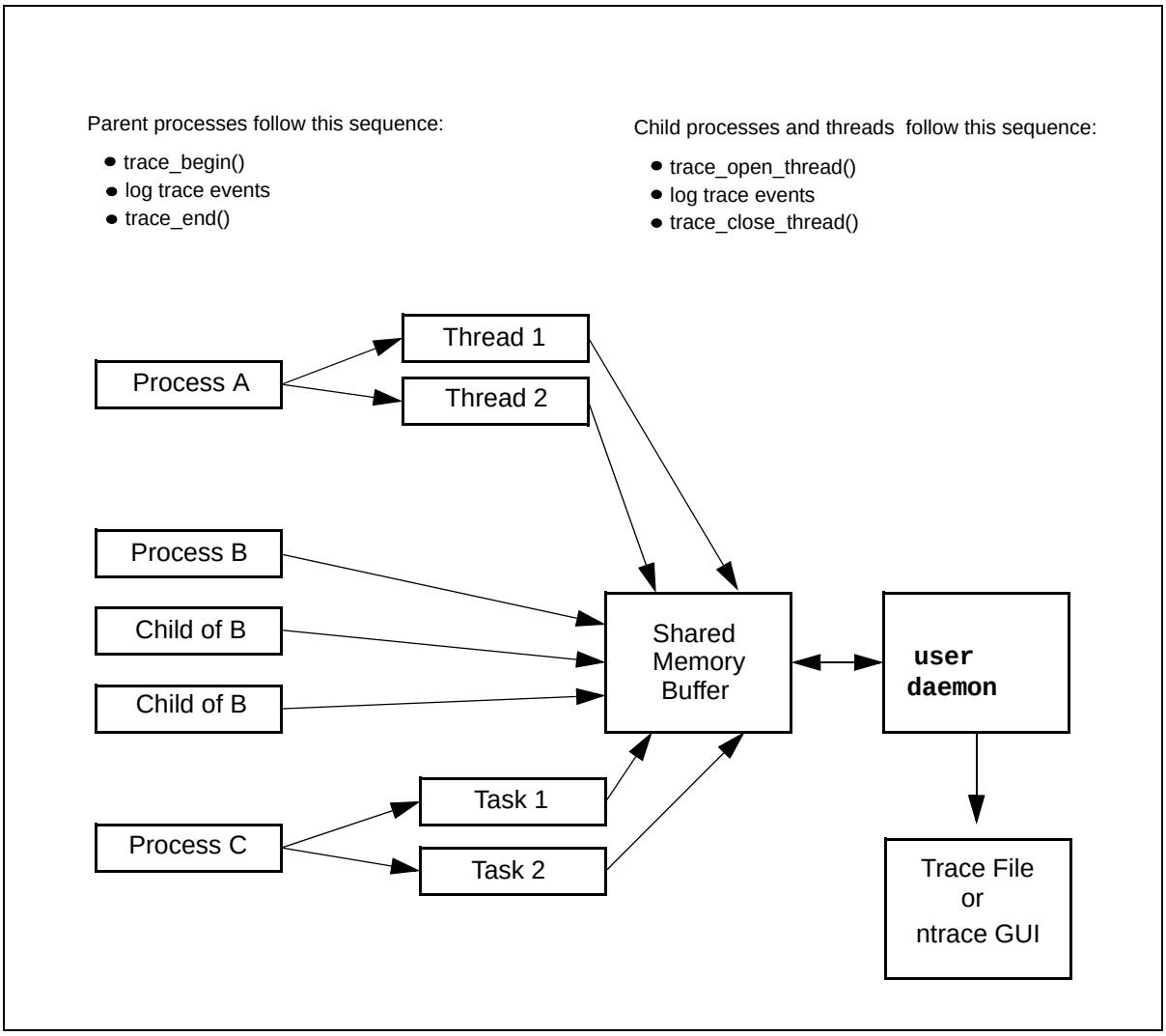


Figure 2-1. Inter-Process Communication and Library Routines

Understanding NightTrace Library Calls

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

- Initialize a tracing session
- Open the current thread for trace event logging
- 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
- Close the current thread for trace event logging
- Terminate a tracing session

The next sections describe these routines in detail.

trace_begin

The `trace_begin` routine initializes the tracing session and acquires resources for your process.

SYNTAX

C:

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

Fortran:

```
integer function trace_begin(trace_file, config)
character *(*) trace_file
integer config(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;
```

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.

config — For C, either a NULL pointer, in which case the default settings are used, or a pointer to a `ntconfig_t` structure.

For 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).

For Fortran, the *config* record must be represented by an array of

NTC_SIZE integer items. Member of the array must be provided as described below.

The following describe the individual parameters:

C: *ntc_version*
Fortran: *config*`(ntc_version)`

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

C: *ntc_lock_pages*
Ada: *lock_pages*
Fortran: *config*`(ntc_lock_pages)`

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*.

C: *ntc_clock*
Ada: *clock*
Fortran: *config*`(ntc_clock)`

Specifies which clock to use as a timing source. This value must be `NT_USE_ARCHITECTURE_CLOCK` or `NT_USE_RCIM_TICK_CLOCK`. The user daemon default value is `NT_USE_ARCHITECTURE_CLOCK`.

C: *ntc_shmid_perm*
Ada: *shmid_perm*
Fortran: *config*`(ntc_shmid_perm)`

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

C: *ntc_daemon_preferred*
Ada: *inherit*
Fortran: *config*`(ntc_daemon_preferred)`

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` call to fail.

C: *ntc_num_buffers, ntc_buffer_length*
Ada: *num_buffers, buffer_length*
Fortran: *config*`(ntc_num_buffers)`, *config*`(ntc_buffer_length)`

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 *ntc_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 length32768.

C: *ntc_daemon_wait_usec*
Fortran: *config*(`ntc_daemon_wait_usec`)

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: *config*(`ntc_reserved`)

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

DESCRIPTION

The `trace_begin` routine performs the following operations:

- Verifies that the version of the NightTrace library linked with the application is compatible with the version used by the user daemon if it is already running
- Verifies the supplied configuration settings are not in conflict with a pre-existing daemon or defines the configuration with these settings if the user daemon does not yet exist.
- Verifies that the RCIM synchronized tick clock is counting if it was selected as the timestamp source
- Attaches the shared memory buffer (after creating it if needed)
- Locks critical NightTrace library routine pages in memory as directed
- Initializes 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`. Processes that result from a fork in a process that has already initialized the tracing session need not call `trace_begin`.

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

RETURN VALUES

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

`[NTNOERROR]` A daemon has already been started that matches the filename passed as *key_file*. The application can begin to log trace events after calling `trace_open_thread`.

`[NTLISTEN]` All operations where successful, but no user daemon matching the filename passed as *key_file* could be found. The application can continue to make NightTrace API calls but attempts to log events will fail until a daemon is started, at which point logging of events will succeed.

`[NTALREADY]` The application has already initialized the trace without an intervening `trace_end`. Tracing can continue in spite of this error. Solution: Remove redundant `trace_begin` calls.

`[NTBADVERSION]` The calling application is linked with the static NightTrace library and the static library is not compatible with the NightTrace library being used by the user daemon. Solution: Relink the application with the static library version which matches the library version being used by the daemon.

`[NTMAPCLOCK]` The selected event timestamp source could not be attached. Solution: If read access is lacking, see your system administrator.

This can also occur if the RCIM synchronized tick clock is selected as the event timestamp source but the tick clock is not counting. Solution: Start the synchronized tick clock by using the **`clock_synchronize(1M)`** command and restart the application.

`[NTPERMISSION]` The calling application lacks permission to attach the shared memory buffer. Solution: Make sure that the same user who started the user daemon is the current user logging trace events in the application.

`[NTPGLOCK]` Permission to lock the text and data pages of the NightTrace library routines was denied. If the user is not privileged to lock pages, see your system administrator or set *ntc_lock_pages* to FALSE.

`[NTNOSHMID]` This can occur if the size of the shared memory buffer exceeds the system limits or the shared memory buffer already exists but the size required by *num_buffers* and *ntc_buffer_length* parameters 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

Related routines include: `trace_open_thread`, `trace_end`

trace_open_thread

The `trace_open_thread` routine associates the current Ada task or C thread with a user-specified name. Use of this library routine is optional. By default, a trace thread context called "main" is associated with the main program. You can override this name by calling `trace_open_thread` from the main program.

SYNTAX

C:

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

Fortran:

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

Ada:

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

PARAMETERS

thread_name

NightTrace's graphical displays and textual summary information indicate which threads logged trace events. If the `trace_open_thread` thread name is null, the **`ntrace`** display utility uses an internal thread ID as a label in these displays.

Naming your threads can make the displays much more readable. `trace_open_thread` lets you associate a meaningful character string name with the current threads' more cryptic numeric 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 must begin with an alphabetic character and consist solely of alphanumeric characters and the underscore. Spaces and punctuation are not valid characters. The following thread names are reserved and cannot be specified to this routine: ALL, NONE.

DESCRIPTION

For multi-threaded applications, C threads and Ada tasks automatically inherit the current thread name of their parent when they are created. You can create additional thread names by calling `trace_open_thread` once per thread or task. Events subsequently generated by these threads or tasks are marked with the specified name, making event analysis much more meaningful.

Important

In order to identify the thread that logged a trace events in multi-threaded applications, you must register your threads with calls to `trace_register_thread` or `trace_open_thread` or create your threads with the `Pthread_create` wrapper routine provided in the **`/usr/lib/libntrace_thr.a`** library. See the description of "Threads and Logging" on page 2-26 for more information.

RETURN VALUES

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

`[NTINIT]` The NightTrace library routines were not initialized or they were initialized but no user daemon has yet been initiated.

Ensure a `trace_begin` call precedes this call. If the preceding `trace_begin` call returned `NTLISTEN`, then a value of `NTINIT` is not a failure condition and once a user daemon is started, subsequent attempts at logging events will succeed.

`[NTINVALID]` An invalid thread name was specified.

`[NT_ALREADY]` The thread-aware version of the NightTrace logging API library, **`libntrace_thr.a`**, was not used when linking or . See the description of "Threads and Logging" on page 2-26 for more information.

SEE ALSO

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

trace_event and Its Variants

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

SYNTAX

C:

```
int trace_event (int ID);

int trace_event_arg (int ID, int arg);
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_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);
```

Fortran:

```
integer function trace_event (ID)
integer ID

integer function trace_event_arg (ID, arg)
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_flt (ID, arg)
integer function trace_event_two_flt (ID, arg1, arg2)
integer ID
real 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;
```

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 6-17 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 6-11 for more information on

event-map files and formats. Alternatively, you could call the `format` function. See “format()” on page 11-106 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 2-17 for more information.

TIP:

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

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

RETURN VALUES

These routines return a zero value (NTNOERROR) on successful completion. Otherwise, they return a non-zero value to identify the error condition. A list of error codes for these routines follows.

[NTINVALID] An invalid trace event ID has been supplied. Solution: Use trace event IDs only in the range 0-4095, inclusive.

[NTINIT] The NightTrace library routines were not initialized or they were initialized but no user daemon has yet been initiated. Ensure a trace_begin call precedes the trace event logging routine call. Once a user daemon is started, subsequent attempts at logging events will succeed.

For multi-threaded applications, if the thread-aware version of the NightTrace logging API library, **libntrace_thr.a**, was used when linking and the calling thread was not created with the Pthread_create NightTrace API call this error will occur and all subsequent attempts to log trace events with this thread will fail. See the description of "Threads and Logging" on page 2-26 for more information.

[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 *ntc_num_buffers* and *ntc_buffer_length* fields of the ntconfig_t configuration buffer passed to trace_begin.

SEE ALSO

Related routines include:

trace_flush, trace_trigger, trace_enable, trace_enable_range, trace_enable_all, trace_disable, trace_disable_range, trace_disable_all

trace_enable, trace_disable, and Their Variants

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

SYNTAX

C:

```
int trace_enable (int ID);

int trace_enable_range (int ID_low, int ID_high);

int trace_enable_all ();

int trace_disable (int ID);

int trace_disable_range (int ID_low, int ID_high);

int trace_disable_all ();
```

Fortran:

```
integer function trace_enable (ID)
integer ID

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

integer function trace_enable_all ()

integer function trace_disable (ID)
integer ID

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

integer function trace_disable_all ()
```

Ada:

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

(procedures)

```
procedure trace_enable (ID : event_type);

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

procedure trace_enable_all;

procedure trace_disable (ID : event_type);
```

```
procedure trace_disable (
D_low : event_type; ID_high : event_type
);

procedure trace_disable_all;
```

(functions)

```
function trace_enable (ID : event_type)
return ntrace_error;

function trace_enable (
ID_low : event_type; ID_high : event_type
)
return ntrace_error;

function trace_enable_all
return ntrace_error;

function trace_disable (ID : event_type)
return ntrace_error;

function trace_disable (
ID_low : event_type; ID_high : event_type
)
return ntrace_error;

function trace_disable_all
return ntrace_error;
```

PARAMETERS

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 and Its Variants" on page 2-12 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` disables one trace event ID. `trace_disable_range` disables a range of trace event IDs, including both range endpoints. `trace_disable_all` disables all trace events. Disabling an already disabled trace event has no effect.

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

TIP:

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

TIP:

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

RETURN VALUES

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

`[NTINIT]`	The NightTrace library routines were not initialized. Solution: Be sure a `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

Related routines include:

`trace_event` and its variants.

trace_flush and trace_trigger

The `trace_flush` and `trace_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;
```

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

When you run in buffer-wraparound mode, you are telling NightTrace to intentionally discard older or less-vital trace events when the shared memory buffer gets full. In buffer-wraparound mode, you must explicitly call `trace_flush` or `trace_trigger`. Only then, does the user daemon copy the remaining trace events from the shared memory buffer to the trace event file. However, do not call `trace_flush` or `trace_trigger` too often or you will reduce the effectiveness

of this mode. See “ntraceud Options” on page 3-3 for more information on buffer-wraparound mode.

RETURN VALUES

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

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

SEE ALSO

Related routines include:

`trace_event` and its variants

trace_close_thread

The `trace_close_thread` routine disables trace event logging for the current thread or process. Use of this routine is not strictly required, unless a subsequent `trace_open_thread`call is desired.

SYNTAX

C: `int trace_close_thread;`

Fortran: `integer function trace_close_thread`

Ada:
```
function trace_close_thread return
ntrace_error;
```

DESCRIPTION

Terminate tracing for the calling thread or Ada task. Subsequent calls to `trace_event` or `trace_event_arg` and its variants will fail unless an intervening call to `trace_open_thread` is made.

RETURN VALUES

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

`[NTINIT]` The NightTrace library routines were not initialized. Solution: Call `trace_close_thread` only once if you previously called `trace_open_thread`.

SEE ALSO

Related routines include:`trace_open_thread`, `trace_end`

trace_end

The `trace_end` routine frees resources and terminates the trace session in your process. Use of this routine 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 this routine is appropriate.

SYNTAX

C:
```
int trace_end;
```

Fortran:
```
integer function trace_end
```

Ada:
```
function trace_end
return ntrace_error;
```

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

RETURN VALUES

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

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

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

SEE ALSO

Related routines include:`trace_begin`, `trace_close_thread`

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`

Ada: `procedure trace_diag_mode;`

DESCRIPTION

Specify a zero value to set the diagnostic mode to FALSE (off) or a non-zero value to set it to TRUE (on).

The NightTrace API diagnostic routine may be changed via the `trace_diag_func` routine.

Additionally, 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

Related routines include `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_mode (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.

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

Related routines include `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 2-17.) 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 distinguis between multiple threads in a multi-threaded application, the following steps must be taken:

1. The application must be linked with the thread-aware version of the NightTrace logging API by specifying the **`-lntrace`**_thr link option.
2. Threads must be registered via calls to `trace_register_thread` or `trace_open_thread` or be created via the `Pthread_create` wrapper function which automatically registers newly created threads.

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

trace_register_thread

The `trace_register_thread` routine registers the calling thread with the NightTrace API

Registration is necessary in order to be able distinguish between threads during event analysis.

SYNTAX

```
#include <ntrace_thr.h>
int trace_register_thread (void);
```

DESCRIPTION

Once registered, the thread's ID can be determined in subsequent event analysis.

In order to associate a textual name with the calling thread, use `trace_open_thread` instead.

Alternatively, threads created using the `Pthread_create` wrapper function are automatically registered.

RETURN VALUES

The `trace_register_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_end` error codes follows.

`[NT_THREAD_ERR]` A failure occurred while attempting to create thread-private data.

SEE ALSO

Related routines include `trace_open_thread and Pthread_create`.

Pthread_create

The `Pthread_create` function is a wrapper around the POSIX **`pthread_create(3)`** function.

This function has the same semantics and syntax as `pthread_create` with the addition that the newly created thread is automaticaly registered via an implicit call to `trace_register_thread`.

SYNTAX

C:
```
int Pthread_create (pthread_t *,
                    pthread_attr_t *,
                    void *(*)(void*),
                    void *);
```

DESCRIPTION

Create a new thread and automatically register it via an implicit call to `trace_register_thread`.

To associate a name with the newly created thread, you must subsequently call `trace_open_thread` from the new thread.

RETURN VALUES

The return values are identical to those defined by **`pthread_create(3).`**

In the unlikely event that the thread registration fails because thread-private data cannot be created, the registration is skipped.

SEE ALSO

Related routines include `trace_open_thread`.

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.

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 C-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 C-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.*

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 "Daemon Definition Dialog" on page 7-57.
- 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
- The default user daemon configuration
- **ntraceud** modes
- **ntraceud** options
- Invoking **ntraceud**

The ntraceud Daemon

When you start up **ntraceud**, it creates a daemon background process and 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 a 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). However, the Real-Time Clock and Interrupt Module (RCIM) can be used as a timestamp source by using the **--rcim** option to **ntraceud** (see "ntracekd Options" on page 4-2).

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.

--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.

--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.

--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.

--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.*

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 topics:

- The **`ntracekd`** daemon
- **`ntracekd`** modes
- **`ntracekd`** options
- Example **`ntracekd`** invocations

The ntracekd Daemon

When you initiate **`ntracekd`**, it creates a daemon background process and returns while that daemon process executes. Once it returns to 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.

--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.

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

This option is required when initiating a daemon and specifies the maximum size of the output file. *size* may be specified as an integer number optionally followed by a **K**, **M**, or **G**, which indicates kilobytes, megabytes, or gigabytes, respectively. If no letter is specified, the units are assumed to be in bytes. *size* may also be **+**, which indicates that the output may grow without limit. Use of **+** is not recommended as kernel tracing can quickly consume vast quantities of disk space.

--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 a 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
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
- Conserving disk space
- Conserving memory and accelerating **`ntrace`**

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:

- From the command line, use the **`run(1)`** command to initiate **`ntraceud`** or **`ntracekd`** with an urgent priority and favorable scheduling policy, e.g.

> **run -P50 -sFIFO ntraceud ...**

- Invoke **ntraceud** and **ntracekd** with the **--priority=***P*, **--policy=***P*, and **--cpu=***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 generated 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.

Part II - Graphical Analysis

Part II Graphical Analysis

6
Invoking NightTrace

NightTrace is invoked using **ntrace** which is normally installed in **/usr/bin**.

The full command syntax for **ntrace** is:

ntrace [**-h**] [**--help**] [**--help-summary**]
[**-v**] [**--version**] [**-l**] [**--listing**]
[**--stats**] [**-n**] [**--notimer**]
[**-s** *val*] [**--start=**{ *offset* | *time*{ **s** | **u** } | *percent***%** }]
[**-e** *val*] [**--end=**{ *offset* | *time*{ **s** | **u** } | *percent***%** }]
[**-hm**] [**--hide-main-window**] [**-***Xoption* ...]
[**-x**] [**--nopages**]
[**-u**] [**--use-session**] [**--summary**=*criteria*]
[**--verbose**]
[**--crash**=*crash_options*]
[*file* ...]

Depending on the options and arguments specified to **ntrace**, NightTrace:

- loads all trace event information into memory
- checks the syntax of specifications in each file argument
- processes each file argument
- loads any display pages and their objects into memory
- presents any display pages (see Chapter 9 “Display Pages”)
- displays the NightTrace Main Window (see Chapter 7 “The NightTrace Main Window”)

Command-line Options

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

-h
--help

Display **ntrace** invocation syntax and a list of all command line options to standard output.

--help-summary

Display help specific to the **--summary** option to standard output.

See “Summary Criteria” on page 6-5 for more information.

-v
--version

Display the current version of NightTrace to standard output and exit.

--crash=*crash_options*

Display 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 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

Display a chronological listing of all trace events and their arguments from all supplied trace-event data files to standard output and exit.

The output includes the following information about a trace event:

- relative timestamp
- trace event ID
- any trace event argument(s)
- the 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 6-11.

NOTE

The CPU field is only meaningful for kernel trace events; for user trace events, the CPU field is displayed as `CPU=??`.

`--stats`

Display simple overall statistics about the trace-event data files to standard output and exit.

The statistics are grouped by trace event file, with cumulative statistics for all trace event files.

The statistics include:

- the number of trace event files
- their names
- the number of trace events logged
- the number of trace events lost

For example, the following command:

```
ntracekd --wait=2 kernel-data
```

collects kernel trace data for two seconds from the system on which it was issued and saves the results to **`kernel-data`** (see Chapter 4 “Capturing Kernel Events with ntracekd”).

Issuing the command:

```
ntrace --stats kernel-data
```

results in the output similar to the following:

```
Read 1 trace event segment timestamped with Intel TSC.
(1)  Kernel trace event log file:  kernel-data.
     226809 trace events plus 204596 continuation events.
     105419 trace events lost.
     2.9707482s time span, from 0.0000000s to 2.9707482s.

     226809 total events read from disk plus 204596 continuation events.
     226808 total events saved in memory; 117 events internal to ntrace.
     105419 total trace events lost.
     2.9707482s total time span saved in memory.
```

Detailed summary information about a trace data set is available via the **`--summary`** option (see page 6-5).

`-n`
`--notimer`

Exclude from analysis trace events for system timer interrupts in the kernel trace file.

`-s` *val*
`--start=`{ *offset* | *time*{ **`s`** | **`u`** } | *percent***`%`** }

Exclude from analysis trace events before the specified trace-event offset, relative time in seconds (**`s`**) or microseconds (**`u`**), or percent of total trace events.

The specified values can be:

offset	Load trace events after the specified trace event offset. (See "Grid" on page 9-29 for information about trace event offsets.)
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***%** }

Exclude from analysis trace events after the specified trace-event offset, relative time in seconds (**s**) or microseconds (**u**), or percent of total trace events.

The specified values can be:

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.

-hm
--hide-main-window

Start NightTrace with the NightTrace Main window hidden (see Chapter 7 "The NightTrace Main Window"); only display pages are shown (see Chapter 9 "Display Pages").

Display pages be subsequently launched using the menu items on the Page menu from on the NightTrace Main window (see "Page" on page 9-3).

-x
--nopages

Start NightTrace with the NightTrace Main Window (see Chapter 7 "The NightTrace Main Window") but do not launch display pages automatically.

The NightTrace Main Window may be subsequently displayed using the NightTrace Main Window... menu item on the Page menu from any display page (see "Pages" on page 7-22).

-u
--use-session

Automatically load the last session used in a previous invocation of NightTrace. All files associated with the previous session are automatically loaded.

--summary=*criteria*

Provide a textual summary of specified trace events using the supplied *criteria*. Summary results are sent to standard output.

See “Summary Criteria” on page 6-5 for details regarding valid *criteria*.

--verbose

In addition to the cumulative statitistics normally output, this option provides detailed information about each occurance of the item being summarized.

-*Xoption ...*

Use any standard X Toolkit command line options. See **X(1)**.

file ...

You can invoke NightTrace with arguments such as trace event files, event map files, page configuration files, session configuration files, or trace data segments.

See “Command-line Arguments” on page 6-10 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 kernel-data
```

Summary Criteria

The **--summary** option is supplied with criteria for command-line usage without ever using the GUI to perform summaries.

NOTE

The **`--verbose`** option (see "`--verbose`" on page 6-5) 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*

Summarize the number of occurrences of the specified *event*.

`p:`*process*

Summarize all events associated with the specified *process*.

`t:`*thread*

Summarize all events associated with the specified *thread*.

`s:`*call*

Summarize all events associated with the entry or resumption of the specified system *call*.

`sl:`*call*

Summarize all events associated with the exit or suspension of the specified system *call*.

`se:`*call*

Summarize all events associated with the specified system *call*.

`ss:`*call*

Summarize all occurrences of a state defined by system call activity for the specified system *call*.

`i:`*intr*

Summarize all events associated with the entry or resumption of the specified interrupt *intr*.

il:*intr*

Summarize all events associated with the exit or interruption of the specified interrupt *intr*.

ie:*intr*

Summarize all events associated with the specified interrupt *intr*.

is:*intr*

Summarize all occurrences of a state defined by interrupt activity for the specified interrupt *intr*.

e:*exc*

Summarize all events associated with the entry or resumption of the specified exception *exc*.

el:*exc*

Summarize all events associated with the exit or interruption of the specified exception *exc*.

ee:*exc*

Summarize all events associated with the specified exception *exc*.

es:*exc*

Summarize all occurrences of a state defined by exception activity for the specified exception *exc*.

skip:on

Suppresses summarization for all subsequent criteria in the list (or until a **skip:off** criteria is seen) if there are no summarization matches for the criteria.

skip:off

Reactivates summarization for all subsequent criteria in the list (or until a **skip:on** criteria is seen) if there are no summarization matches for the criteria.

st:*start***-***end*

Summarize all occurrences of the state defined by the starting event *start* and terminated by the ending event *end*.

These may be combined together along with tagged criteria from the Summarize NightTrace Events dialog (see "Summarizing Statistical Information" on page 8-19) in a comma-separated list.

Consider the following example:

```
ntrace --summary=ss:read,ss:alarm,ev:5,crit_0 event_file my_session
```

Using the trace event file **`event_file`** as the trace data source (see "Trace Event Files" on page 6-11), NightTrace will:

1. summarize the number of occurrences of `read` and `alarm` system call states that occur in the data source; provide information pertaining to the duration of each state (`min`, `max`, `avg`, `sum`); and provide information related to the gaps between each state (`min`, `max`, `avg`, `sum`)
2. summarize the number of occurences of user events with a *trace event ID* of 5 as well as information about the gaps between the events (`min`, `max`, `avg`)
3. perform a summary using the profile defined by `crit_0` in the **`my_session`** session file (see "Session Configuration Files" on page 6-24)

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 6-24).

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 6-11 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 6-11 for more information.

- page configuration files

  Configuration files define display pages, the display objects contained within them, string tables, and format tables. These ASCII files are usually created by NightTrace.

  See "Configuration Files" on page 6-14 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 6-24 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 NightTrace menu on the NightTrace Main Window (see "Save Trace Segments..." on page 7-5).

  See "Trace Data Segments" on page 6-25 for more information.

Trace Event Files

Trace event files are created by user and kernel trace daemons. They consist of header information and individual trace events and their arguments as logged by user applications or the operating system. NightTrace detects trace event files as specified on the command line and does the required initialization processing so that the trace events contained in the files are available for display.

To load a trace event file, either:

- specify the trace event file as an argument to the **`ntrace`** command when you invoke NightTrace
- Select the Open Trace File… menu option from the NightTrace menu of the NightTrace Main window (see "NightTrace" on page 7-2) and select the trace event file from the file selection dialog

Event Map Files

NightTrace does not require you to use event map files. However, if you use these file(s), you can improve the readability of your NightTrace displays.

An *event map file* allows you to associate meaningful names with the more cryptic trace event ID numbers. It also allows you to associate additional information with a trace event including the number of arguments and the argument conversion specifications or display formats. Although NightTrace does not require you to use event map files, labels and display formats can make graphical NightTrace displays and textual summary information much more readable.

To load an existing event map file, perform any of the following:

- specify the event map file as an argument to the **`ntrace`** command when you invoke NightTrace
- select the Open Event Map File… menu item from the NightTrace menu on the NightTrace Main Window (see "Open Event Map File..." on page 7-5)

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 11-107.

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 specifica-

tion 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.)

You may also add or edit event map entries by selecting the Event Maps… menu option from the Edit menu of the NightTrace Main window.

For more information on event map files, see “Pre-Defined Strings Tables” on page 6-17.

Configuration Files

A *configuration file* contains information related to the layout of a particular display page and includes the configurations of all display objects that have been created on that page. In addition, any user-defined tables that have been created for that page is also contained in this file. Although NightTrace does not require you to use page configuration files, using a page configuration file improves the readability of your display pages and saves you time laying out your display pages.

A page configuration file is an ASCII file containing such definitions as:

- display page definitions (see Chapter 9 "Display Pages")
- string table definitions (see "String Tables" on page 6-16)
- format table definitions (see "Format Tables" on page 6-20)

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.

You can create, modify, save, and load configuration files from within NightTrace; however, you must use a text editor to create and modify tables in a configuration file. NightTrace ignores blank lines and treats text between a `/*` and a `*/` as comments in configuration files; however, saving a configuration file removes your comments.

To load an existing configuration file, either:

- specify the configuration file as an argument to the **`ntrace`** command when you invoke NightTrace
- Select the Open Config File... menu option from the NightTrace menu of the NightTrace Main window and select the configuration file from the file selection dialog

Tables

The page configuration file (see "Configuration Files" on page 6-14) may contain two types of tables, both of which can improve the readability of your NightTrace displays:

- string tables (see "String Tables" on page 6-16)

- format tables (see “Format Tables” on page 6-20)

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 6-17, and page 12-14.

If you define a string table or format table more than once in a configuration file, NightTrace merges the two tables; if there are duplicate entries, values come from the last definition.

String Tables

You can log a trace event with one or more numeric arguments. Sometimes these arguments can take on a nearly fixed set of values. A *string table* associates an integer value with a character string. Labeling numeric values with text can make the values easier to interpret.

The syntax for a string table is:

```
string_table ( table_name ) = {
   item = int_const, "str_const" ;
   ...
   [ default_item = "str_const" ; ]
};
```

Include all special characters from the syntax except the ellipsis (`...`) and square brackets (`[]`).

The fields in a string table definition are:

`string_table`

The keyword that starts the definition of all string tables.

table_name

The unique, user-defined name of this table. This name describes the relationship of the numeric values in this string table.

An *item line* associates an integer value with a character string. This line extends from the keyword `item` through the ending semicolon. You may define any number of item lines in a single string table. The fields in an item line are:

`item`

The keyword that begins all item lines.

int_const

An integer constant that is unique within *table_name*. It may be decimal, octal, or hexadecimal. Decimal values have no special prefix. Octal values begin with a zero (`0`). Hexadecimal values begin with `0x`.

str_const

A character string to be associated with *int_const*. Keep this string short; otherwise, NightTrace may be unable to display it in the limited window space available. Use a `\n` for a newline, not a carriage return in the middle of the string.

The optional *default item line* associates all other integer values (those not explicitly referenced) with a single string.

TIP

If your table needs only one entry, you may omit the item line and supply only the default item line. A `get_string()` call with this table name as the first parameter needs no second parameter.

NightTrace returns a string of the item number in decimal if:

- there is no default item line, and the specified item is not found
- the string table is not found (The first time NightTrace cannot find a particular string table, NightTrace flags it as an error.)

The following lines provide an example of a string table in a configuration file.

```
string_table (curr_state) = {
   item =  3, "Processing Data";
   item =  1, "Initializing";
   item = 99, "Terminating";
   default_item = "Other";
};
```

In this example, your application logs a trace event with a numeric argument that identifies the current state (`curr_state`). This argument has three significant values (`3`, `1`, and `99`). When `curr_state` has the value `3`, the NightTrace display shows the string "`Processing Data`." When it has the value `1`, the display shows "`Initializing`." When it has the value `99`, the display shows "`Terminating`." For all other numeric values, the display shows "`Other`."

For more information on string tables and the `get_string()` function, see page 11-100.

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.

NOTE

When analyzing trace event files from multiple systems, thread identifiers are not guaranteed to be unique across nodes. Therefore, accessing the `tid` table may result in an incorrect thread name being returned for a particular thread ID. To get the correct thread name for a thread ID, the `tid` table for the node on which the process identifier occurs should be used instead. The `tid` table is maintained for backwards compatibility.

This table is indexed by a thread identifier or a thread name. Examples of using this table are:

```
get_string(tid, tid())
get_item(tid, "cleanup_thread")
```

`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 11-31 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 12-14.

`syscall`

See page 12-14.

`device`

See page 12-14.

`vector_`*nodename*

See page 12-14.

`syscall_`*nodename*

See page 12-14.

`device_`*nodename*

See page 12-14.

You can use pre-defined string tables anywhere that string tables are appropriate. Use the `get_string()` function to look up values in string tables. For information about the `get_string()` function, see page 11-100.

Format Tables

Like string tables, *format tables* let you associate an integer value with a character string; however, in contrast to a string table string, a format table string may be dynamically formatted and generated. Labeling numeric values with text can make the values easier to interpret.

The syntax for a format table is:

```
format_table ( table_name ) = {
   item = int_const, "format_string" [ , "value1" ] ... ;
   ...
   [ default_item = "format_string" [ , "value1" ] ... ; ]
};
```

Include all special characters from the syntax except the ellipses (`...`) and square brackets (`[]`).

The fields in a format table are:

`format_table`

The keyword that begins the definition of all format tables.

table_name

The unique, user-defined name of this table. This name describes the relationship of the numeric values in this format table.

An *item line* associates a single integer value with a character string. This line extends from the keyword `item` through the ending semicolon. You may have any number of item lines in a single format table.

The fields in an item line are:

`item`

The keyword that begins all item lines.

int_const

An integer constant that is unique within *table_name*. This value may be decimal, octal, or hexadecimal. Decimal values have no special prefix. Octal values begin with a zero (`0`). Hexadecimal values begin with `0x`.

format_string

A character string to be associated with *int_const*. Keep this string short; otherwise, NightTrace may be unable to display it in the limited window space available. Use a `\n` for a newline, not a carriage return in the middle of the string.

The string contains zero or more conversion specifications or display formats. Valid conversion specifications for displays include the following:

`%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 values. A particular conversion specification-quoted value pair must match in both data type and position. For example, if *format_string* contains a `%s` and a `%d`, the first quoted value must be of type string and the second one must be of type integer. If the number or data type of the quoted value(s) do not match *format_string*, the results are not defined.

value1

A value associated with the first conversion specification in *format_string*. The value may be a constant string (literal) expression or a NightTrace expression. A string literal expression must be enclosed in double quotes. An expression may be a `get_string()` call (see page 11-100). For more information on expressions, see Chapter 11 "Using Expressions".

The optional `default_item` line associates all other integer values with a single format item. NightTrace flags it as an error if an expression evaluates to a value that is not on an item line and you omit the default item line.

TIP

If your table needs only one entry, you may omit the item line and supply only the default item line. A `get_format()` call with this table name as the first parameter needs no second parameter.

The following lines provide an example of a string table and format table in a configuration file.

```
string_table (curr_state) = {
   item =  3, “Processing Data”;
   item =  1, “Initializing”;
   item = 99, “Terminating”;
   default_item = “Other”;
};

format_table (event_info) = {
   item = 186, “Search for the next time we process data”;
   item =  25, “The current state is %s”,
               “get_string (curr_state, arg1())”;
   item = 999, “Current state is %s, current trace event is %d”,
               “get_string (curr_state, arg1())”,
               “offset()”;
   default_item = “Other”;
};
```

In this example, the first numeric argument associated with a trace event represents the current state (`curr_state`), and the `event_info` format table represents information associated with the trace event IDs. When trace event `186` occurs, a `get_format(event_info,186)` makes NightTrace display:

```
Search for the next time we process data
```

When trace event `25` occurs, NightTrace replaces the conversion specification (`%s`) with the result of the `get_string()` call. If `arg1()` has the value 1, then NightTrace displays:

```
The current state is Initializing
```

When trace event `999` occurs, NightTrace replaces the first conversion specification (`%s`) with the result of the `get_string()` call and replaces the second conversion specification (`%d`) with the integer result of the numeric expression `offset()`. If `arg(1)` has the value 99 and `offset()` has the value 10, then NightTrace displays:

```
Current state is Terminating, current trace event is
10
```

For all other trace events, NightTrace displays “`Other`”.

For more information on `get_string()`, see “get_string()” on page 11-100.

For more information on format tables and the `get_format()` function, see “get_format()” on page 11-104.

For more information about `arg1()`, see “arg()” on page 11-16.

For more information about `offset()`, see “offset()” on page 11-25.

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 6-5) is strongly encouraged to invoke NightTrace with the last session loaded or saved.

A session configuration may include:

- daemon definitions

  See "Daemon Definition Dialog" on page 7-57 for more information.

- display page configurations

  See "Configuration Files" on page 6-14 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 11-1 for more information.

- named tags

  See "Tags..." on page 9-13 for more information.

- previously-executed searches

  See "Search/Close" on page 8-17 for more information.

- previously-executed summaries

  See "Summarizing Statistical Information" on page 8-19 for more information.

- references to saved trace data segment files

  See "Trace Data Segments" on page 6-25 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 “Kernel” on page 7-63)
- 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 “User Application” on page 7-63)

Session configuration files can be generated by the following menu items in the NightTrace menu of the NightTrace Main Window:

- Save Session (see “Save Session” on page 7-4)
- Save Session Copy (see “Save Session Copy” on page 7-5)
- Save Session As… (see “Save Session As...” on page 7-4)

Upon exiting when there are unsaved changes to the session, the user is given the chance to Save Session and Exit or Save Session Copy and Exit. See “Unsaved Changes” on page 7-7.

The user may load the session on a subsequent invocation of NightTrace by either:

- specifying the session configuration filename on the command-line when invoking **ntrace** (see “Invoking NightTrace” on page 6-1)
- using the Open Session dialog (see “Open Session...” on page 7-3) 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 Segments… menu option from the NightTrace menu on the NightTrace Main window (see “NightTrace” on page 7-2).

7
The NightTrace Main Window

The NightTrace GUI is invoked using **`ntrace`** (see “Invoking NightTrace” on page 6-1).

By default, the NightTrace Main window is presented as shown in the figure below.

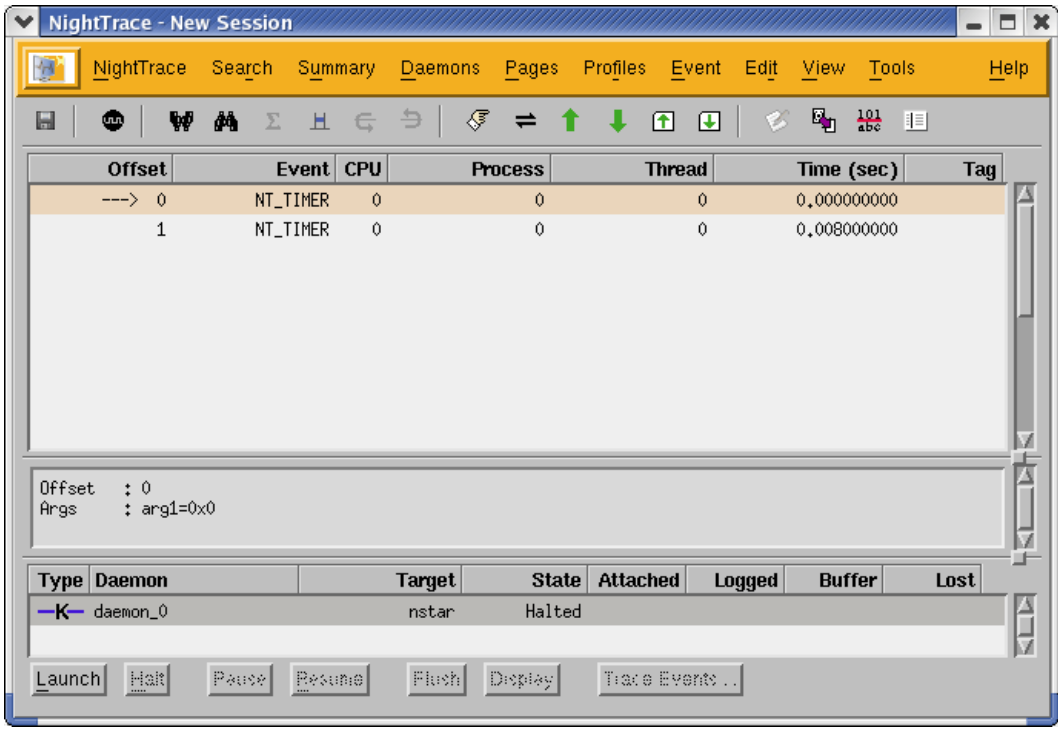


Figure 7-1. NightTrace Main Window

The NightTrace Main window consists of the following components:

- NightTrace Main window Menu Bar (see page 7-2)
- NightTrace Main window Tool Bar (see page 7-43)
- Profile Area (see 7-45)
- Event Display Areas (see 7-46)
- Trace Segment Statistic Area (see 7-48)
- Daemon Control Area (see page 7-50)

NightTrace Main window Menu Bar

The NightTrace Main window menu bar is a part of the NightTrace Main window (see "The NightTrace Main Window" on page 7-1).

The NightTrace Main window menu bar provides access to the following menus:

- NightTrace
- Search
- Summary
- Daemons
- Pages
- Profiles
- Event
- Edit
- View
- Tools
- Help

Each menu is described in the sections that follow.

NightTrace

The NightTrace menu contains *session*-related items such as initiating a new *session*, saving the current session to a configuration file, and opening a previously-saved configuration file.

A session includes daemon configurations, trace data sets, configuration options, display pages, and user-defined profiles.

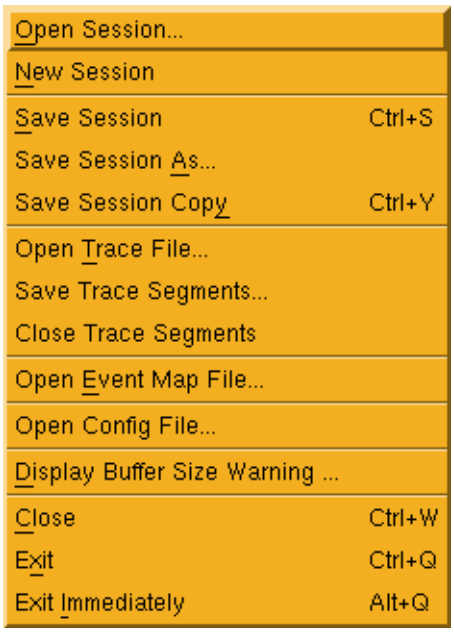


Figure 7-2. NightTrace menu

Open Session...

Displays the Open Session dialog allowing the user to navigate to the desired directory and select a previously-saved session configuration file to open (see "Session Configuration Files" on page 6-24).

NOTE

Filenames are relative to the *host system* (the system where the NightTrace GUI is running) in the Open Session dialog.

NOTE

NightTrace will automatically load the last session used when invoked with the **-u** option. See "Invoking NightTrace" on page 6-1 for more information.

If an attempt is made to open a previously-saved session configuration file when changes have been made to the current configuration but have not yet been saved, the Unsaved Changes dialog is presented to the user (see "Unsaved Changes" on page 7-7).

New Session

Creates a new *session*.

If an existing session is open, it is first closed by this operation.

If changes have been made to the current configuration but have not yet been saved, the Unsaved Changes dialog is presented to the user (see "Unsaved Changes" on page 7-7).

Save Session

Save Session saves the current session to a session configuration file (see "Session Configuration Files" on page 6-24 for a complete description of the contents of a session).

Save Session allows for quickly saving a session. The user is not prompted for the filenames where the session, trace data, or display pages are to be saved. These are automatically saved in appropriately named files in the current working directory.

If the current session has not been saved to a file in the past, the session is automatically saved to a new session configuration file. The new filename appears in the 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 (see "Discard Events..." on page 9-12). 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...

Displays the Save Session dialog allowing the user to navigate to the desired directory and specify the name of the file to which the session configuration will be saved (see "Session Configuration Files" on page 6-24).

NOTE

Filenames are relative to the *host system* (the system where the NightTrace GUI is running) in the Save Session dialog.

The new filename appears in the Trace Segment Statistics area above the Daemon Control Area in the NightTrace Main window (see “Trace Segment Statistic Area” on page 7-48).

Save Session Copy

Save Session Copy saves the current session to a newly created session configuration file (see “Session Configuration Files” on page 6-24 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 comon 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.

Open Trace File...

Presents the user with a standard file selection dialog so that they may select a trace event file to load. The event file can be a user trace data file or a kernel trace data file.

Save Trace Segments...

Allows the user to select a filename for saving the segments. Multiple segments may be selected and will be combined into a single segment when written to the specified file.

Close Trace Segments

Closes the trace data segments currently selected in the Trace Segment Statistic 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.

Open Event Map File...

Presents the user with a standard file selection dialog to select an event map file to load. An event map file provides ASCII names for specific trace event values.

See “Event Map Files” on page 6-11 for more information.

Open Config File...

Presents the user with a standard file selection dialog to select a configuration file to load. Configuration files contain string and format tables as well as display page definitions.

See “Configuration Files” on page 6-14 for more information.

Display Buffer Size Warning...

Allows the user to select the memory size threshold for warnings when **`ntrace`** display buffers exceed the specified size and whether **`ntrace`** should automatically halt any active daemons when the threshold is exceeded.

Close

Closes the NightTrace Main window but leaves NightTrace running if other windows are active.

If no other windows are active and if changes have been made to the current configuration but have not yet been saved, the Unsaved Changes dialog is presented to the user (see "Unsaved Changes" on page 7-7).

Exit

Closes the session and exits NightTrace completely.

If changes have been made to the current configuration but have not yet been saved, the Unsaved Changes dialog is presented to the user (see "Unsaved Changes" on page 7-7).

Exit Immediately

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

Unsaved Changes

The Unsaved Changes dialog is presented whenever the user attempts to close the session without saving changes that have been made.

Figure 7-3 shows the dialog that is presented whenever the user attempts to exit NightTrace without saving changes that have been made.

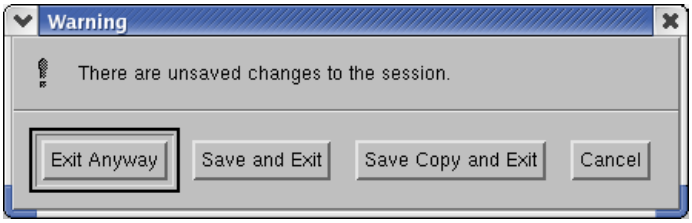


Figure 7-3. Unsaved Changes / Exit dialog

NOTE

To avoid the warning concerning unsaved changes to NightTrace when exiting, the user may chose the Exit Immediately menu item (see "Exit Immediately" on page 7-6). However, any unsaved changes will be lost.

Exit Anyway

Discard the unsaved changes and exit NightTrace.

Save and Exit

Save the unsaved changes and exit.

If the session had not been saved to a session configuration file (see "Session Configuration Files" on page 6-24) before, the name of the newly created session configuration file is presented in a dialog before exiting.

Save Copy and Exit

Save the entire session, along with unsaved changes, to a new session configuration file (see "Session Configuration Files" on page 6-24).

Trace data is saved to a newly created trace data file associated with the session (see "Save Session Copy" on page 7-5 for more details).

The name of the newly created session configuration file is presented in a dialog before exiting.

Cancel

Aborts the exit request.

Figure 7-4 shows the dialog that is presented whenever the user attempts to close the current session without exiting (performing a session operation such as New Session, Open Session, or E xit).

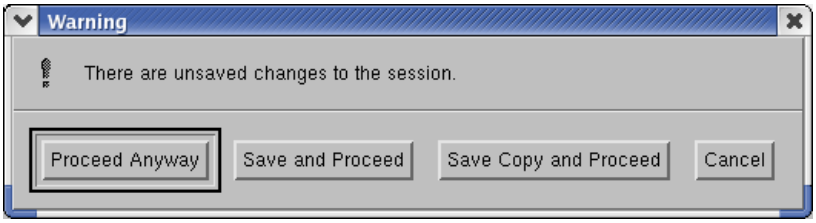


Figure 7-4. Unsaved Changes / Proceed dialog

Proceed Anyway

Discard the unsaved changes and proceed with session operation.

Save and Proceed

Save the unsaved changes and proceed with the session operation.

If the session had not been saved to a session configuration file (see "Session Configuration Files" on page 6-24) before, the name of the newly created session configuration file is presented in a dialog before proceeding.

Save Copy and Proceed

Save the entire session, along with unsaved changes, to a new session configuration file (see "Session Configuration Files" on page 6-24).

Trace data is saved to a newly created trace data file associated with the session (see "Save Session Copy" on page 7-5 for more details).

The name of the newly created session configuration file is presented in a dialog before proceeding.

Cancel

Aborts the session operation.

Search

The Search menu contains search-related items such as opening the Search diaglog to define search criteria, executing a forward or backward search with the most recent search criteria, or modifying search options.

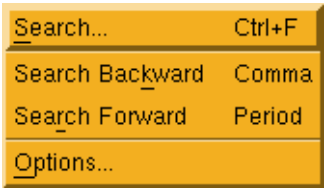


Figure 7-5. Search menu

Search...

Accelerator: Ctrl+F

Displays the Profiles dialog allowing the user to define the search criteria and to execute a search. See “Profiles” on page 8-1 for more information.

Search Forward

Accelerator: Period

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

Accelerator: Comma

Executes a forward search using the last profile defined or selected. If no profiles have been defined, a backward search for the previous event is executed.

Options...

Displays the Search Options dialog as shown below:

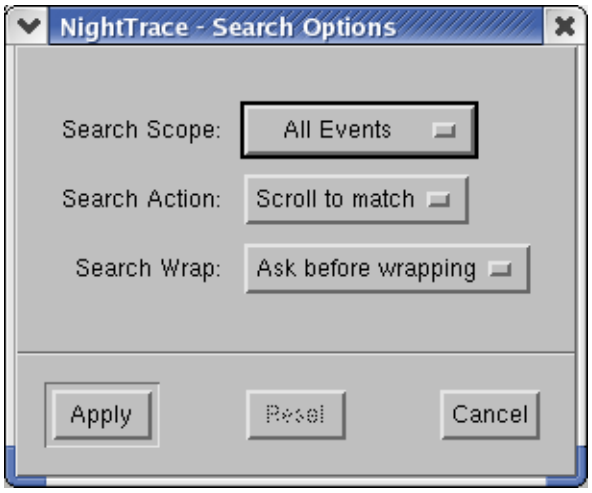


Figure 7-6. Search Options dialog

Search Scope

This option list allows you to define the scope of searches.

All Events

Sets the scope to the entire data set.

Current Interval

Sets the scope to the events included by the current time interval defined by the Start Time and End Time settings in the Interval Control area of display pages. See Section "Interval Control Area" on page 9-32 for more information.

Search Action

This option list controls the actions taken when a search is executed

Scroll to match

Causes the current timeline to be moved to the event matching the search criteria.

Zoom to match

Moves the current timeline and additionally expands the current interval to include the event matching the search criteria.

Do nothing

Prevents the current timeline from moving, but the search results are still printed in the text display areas of the NightTrace Main window and in display pages.

Search Wrap

This option list controls whether a search should wrap around to the other end of the data set when either the beginning or end is reached.

Ask before wrapping

Causes a dialog to pop up when either end of the data set is reached and allows you to continue searching at the other end or to cancel the search.

Wrap around

Automatically causes wrap-around searching.

Do not wrap

Causes searches to fail if they encounter the beginning or end of the data set.

Summary

The Summary menu provides for defining profiles for summaries, executing summaries, and controlling summary options.

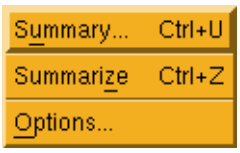


Figure 7-7. Summary menu

Summary...

Accelerator: Ctrl+U

Opens the Profiles dialog (see “Profiles” on page 8-1) allowing the user to select a profile to summarize or define a new profile to summarize.

Summarize

Accelerator: Ctrl+Z

Executes a summary on the current profile. If no profiles has been defined, a summary of all events is executed.

Options...

Opens the Summary Options dialog which controls how summaries operate

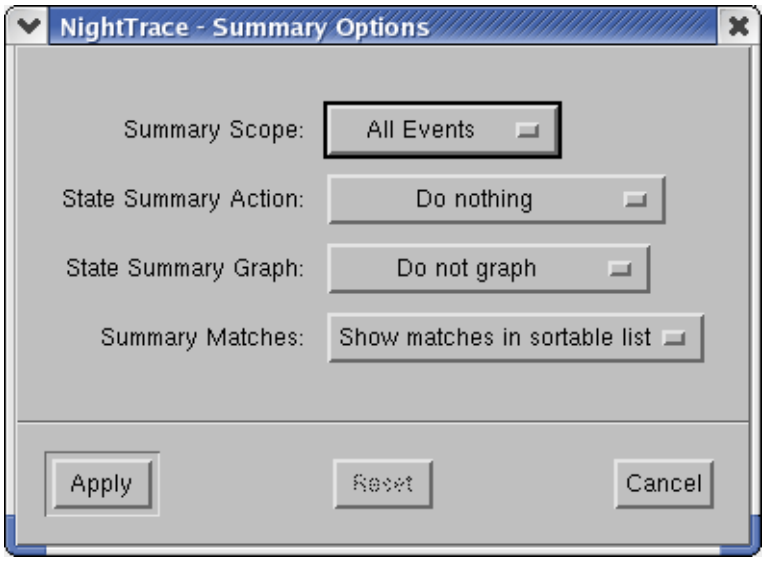


Figure 7-8. Summary Options dialog

Summary Scope

This option list allows you to define the scope of summaries.

All Events

Sets the scope to the entire data set.

Current Interval

Sets the scope to the events included by the current time interval defined by the Start Time and End Time settings in the Interval Control area of display pages. See Section "Interval Control Area" on page 9-32 for more information.

Region

Sets the scope to the selected region defined by the current Mark and the current timeline. See Section "Set the Mark to the current timeline" on page 9-27 for more information.

State Summary Action

This option list controls whether the current timeline moves when a summary is executed. These options are only relevant for summaries of profiles which defined a state.

Scroll to longest duration

Moves the current timeline to the beginning of the state with the longest duration within the summary scope.

Scroll to shortest duration

Moves the current timeline to the beginning of the state with the shortest duration within the summary scope.

Do nothing

Prevents the current timeline from being moved, but the summary results are still displayed in page text areas.

State Summary Graph

Display a Data Graph (see "Data Graph" on page 10-8) showing either the durations of each state on which the summary is based or the gaps between the states.

NOTE

The scale factor for these graphs is automatically determined by the shortest and longest values found. This can sometimes have the effect of obscuring useful data. Consider a situation where 99% of the state instances had a duration on the order of 10-30 microseconds, but a single instance lasted 500000 microseconds. The resulting graph would have a single large spike with the details of the remaining states difficult to ascertain. Use the (*n* x Std. Dev.) menu items in such instances.

The user may select one of the following choices from the drop-down menu:

Durations

Display a Data Graph showing the durations of each state on which the summary is based.

Durations -1 x Std. Dev.

Display a Data Graph showing the durations of each state on which the summary is based.

The scale factor for the graph is automatically determined by the minimum and maximum state duration that fall within one standard deviation of the actual minimum and maximum. All state durations will appear on the graph.

Durations - 2 x Std. Dev.

Display a Data Graph showing the durations of each state on which the summary is based.

The scale factor for the graph is automatically determined by the minimum and maximum state duration that fall within two standard deviations of the actual minimum and maximum. All state durations will appear on the graph.

Gaps

Display a Data Graph showing the durations of the gap between the states on which the summary is based.

Gaps - 1 x Std. Dev.

Display a Data Graph showing the durations of the gap between the states on which the summary is based.

The scale factor for the graph is automatically determined by the minimum and maximum duration of the gaps between states that fall within one standard deviation of the actual minimum and maximum. All state durations will appear on the graph.

Gaps - 2 x Std. Dev.

Display a Data Graph showing the durations of the gap between the states on which the summary is based.

The scale factor for the graph is automatically determined by the minimum and maximum duration of the gaps between states that fall within two standard deviations of the actual minimum and maximum. All state durations will appear on the graph.

Do not graph

No state summary graph is displayed.

Summary Matches

This option list controls whether an additional summary results dialog is shown when summaries are executed.

Show matches in sortable list

Causes the dialog to appear when a summary is executed. The dialog contains sortable columns. Clicking on a column header causes the entries to be sorted by that criteria Subsequent clicks on the same column header reverse the sort order. For state summaries, the Start Offset, End Offset, Gap, and Duration of each state in the summary scope are displayed. For other summaries, the Offset and Gap columns are shown. Selecting a row in the list causes the current timeline to move to event associated with that row (for states, the event that defines the start of the state).

Do not show matches

Prevents the summary results dialog from being displayed.

Daemons

The Daemons menu provides functionality for configuring new and existing daemon definitions, as well as attaching to and detaching from running daemons.

Figure 7-9. Daemons menu

New...

Accelerator: Ctrl+D

Opens the Daemon Definition dialog (see “Daemon Definition Dialog” on page 7-57) allowing the user to configure a new daemon definition.

Edit...

Opens the Daemon Definition dialog (see “Daemon Definition Dialog” on page 7-57) for the daemon definition currently selected in the Daemon Control Area (see “Daemon Control Area” on page 7-50) allowing the user to edit that particular definition.

NOTE

The daemon definition may not be altered while the daemon is executing.

Delete

Deletes the daemon definition(s) currently selected in the Daemon Control Area (see "Daemon Control Area" on page 7-50).

The user is prompted for confirmation before the deletion is performed.

Launch

Accelerator: Ctrl+L

Starts execution of the daemon(s) currently selected in the Daemon Control Area.

NOTE

Starting a daemon does not imply that the daemon begins to collect events.

Launch operations are time consuming and involve possibly connecting to a target system, user authentication, etc. Once the daemon is launched, it is more efficient to utilize the Pause and Resume operations which require less time and resources.

The same action is performed by pressing the Launch button in the Daemon Control Area (see "Launch" on page 7-52).

Halt

Accelerator: Ctrl+H

Stops execution of the daemon(s) currently selected in the Daemon Control Area.

The connection to the target system is terminated by this operation. Once the daemon is launched, it may be more efficient to utilize the Pause and Resume operations.

The same action is performed by pressing the Halt button in the Daemon Control Area (see "Halt" on page 7-53).

Pause

Pauses the execution of the daemon(s) currently selected in the Daemon Control Area.

NOTE

When a daemon is paused, incoming trace events are discarded without notice.

The same action is performed by pressing the Pause button in the Daemon Control Area (see "Pause" on page 7-53).

Resume

Accelerator: Ctrl+R

Resumes execution of the daemon(s) currently selected in the Daemon Control Area. Once resumed, incoming events are placed into the daemon buffer for subsequent processing by the daemon.

The same action is performed by pressing the Resume button in the Daemon Control Area (see "Resume" on page 7-53).

Flush Buffer

Accelerator: Ctrl+F

Flushes trace events from the buffers associated with the daemon(s) currently selected in the Daemon Control Area to either the NightTrace display buffer (see "Stream" on page 7-64) or to the output file (see "Output File" on page 7-65).

The same action is performed by pressing the Flush button in the Daemon Control Area (see "Flush" on page 7-53).

Attach...

Allows the user to query any target system for user application trace daemons and displays the results in the Attach Daemons dialog (see "Attach Daemons" on page 7-20). The user may then attach to the desired daemon and control it.

Detach

Relinquishes control of the running daemon(s) currently selected in the Daemon Control Area (see "Daemon Control Area" on page 7-50).

Reset

Flushes the contents of trace buffers for the running daemon(s) currently selected in the Daemon Control Area (see "Daemon Control Area" on page 7-50). Any events in the buffer at the time of the reset are discarded. Events that have already been written to the output device (file or stream) are unaffected.

Pressing the Reset button also places the selected daemons in a Paused state (see "State" on page 7-51).

NOTE

This option is not supported for kernel trace daemons.

Refresh Rate...

Provides a dialog which controls the refresh interval of statistics for active daemons.

Login

This dialog is presented when attaching to a daemon on a remote system (see "Attach Daemons" on page 7-20).

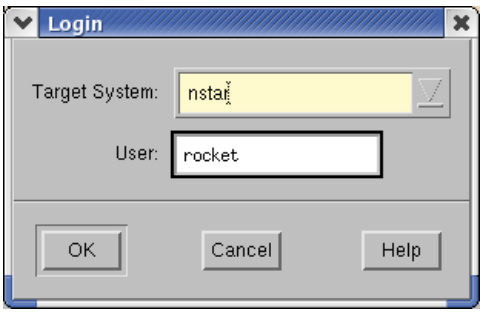


Figure 7-10. Login dialog

After filling in the required fields in the Login dialog, the Enter Password dialog (see "Enter Password" on page 7-18) is displayed, allowing the user to enter the password for the specified User on the specified Target System.

NOTE

Passwords are not included in the configuration files written by NightTrace. They are retained only during the current invocation of NightTrace.

Target System

The name of the target system to which the user wishes to connect.

User

The login name of the user on the specified Target System.

Enter Password

The Enter Password dialog is displayed during user authentication on a target system.

NOTE

The Enter Password dialog is not displayed if a valid password has already been entered for the specified user on the specified target system during the current invocation of NightTrace.

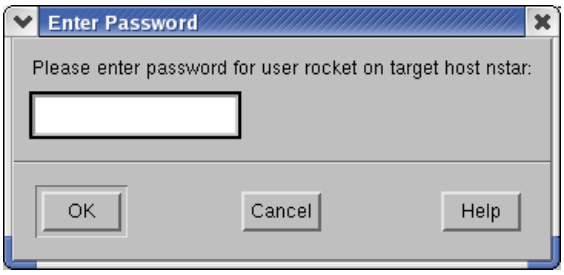


Figure 7-11. Enter Password dialog

Enter the password for the specified user on the specified target system.

NOTE

Passwords are not included in the configuration files written by NightTrace. They are retained only during the current invocation of NightTrace. Passwords are encrypted before being transmitted to the target system for user authentication.

Attach Daemons

The Attach Daemons dialog is displayed when the user attempts to attach to a daemon running on a remote target system.

This dialog is presented following user authentication (see "Login" on page 7-18 and "Enter Password" on page 7-18) on that system.

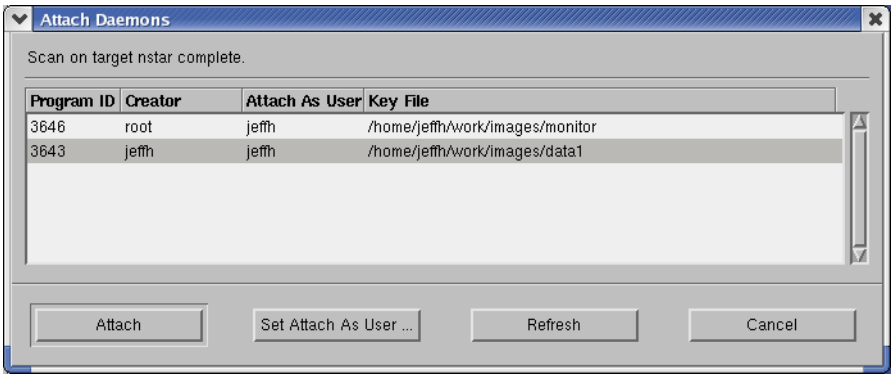


Figure 7-12. Attach Daemons dialog

Program ID

The process ID (PID) of the user trace daemon on the remote system.

Creator

The login name of the user who owns the user trace daemon on the remote system.

Attach as User

The login name of the user attaching to the user trace daemon. This value defaults to the user specified in the Login dialog (see "Login" on page 7-18) presented prior to this dialog.

Key File

The filename which is used to calculate the shared memory segment identifier associated with the logging of user trace events. See "Key File" on page 7-64 for more information.

The following buttons appear at the bottom of the Attach Daemons dialog and have the specified meaning:

Attach

Attaches to the daemon selected in the list and closes the Attach Daemons dialog.

Set Attach as User...

Brings up a dialog allowing the user to specify the login name used to attach to the selected daemon(s). Since the daemon's shared memory is owned by the creator, the user attaching to the user trace daemon could be relevant in terms of permissions.

Refresh

Queries the target system for active trace daemons.

Cancel

Closes the Attach Daemons dialog without attaching to any of the listed daemons.

Pages

The Pages menu allows the user to open pre-configured display pages as well as empty display pages. There is also an option for the user to open up a pre-existing display page.

The Pages menu appears on the NightTrace Main window menu bar (see "NightTrace Main window Menu Bar" on page 7-2).

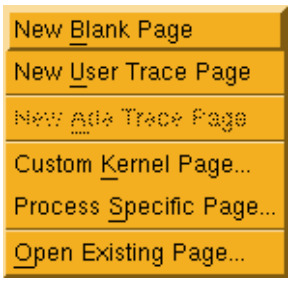


Figure 7-13. Pages menu

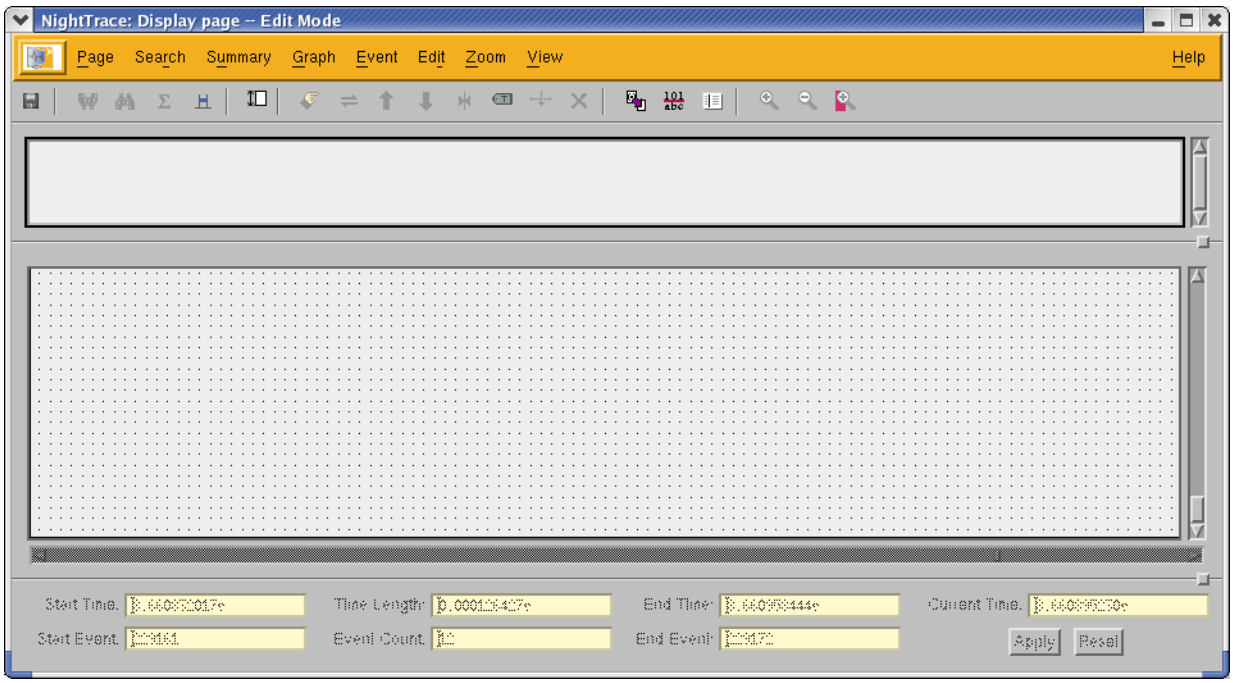
New Blank Page

This menu choice opens a new display page (see Chapter 9 "Display Pages") so that the user may configure it from scratch. The Grid (see must be populated with display objects (see Chapter 10 "Display Objects") before trace information can be analyzed or graphically examined.

NOTE

The new display page comes up in *edit mode* so that display objects may be created and configured (see “Switch between view and edit mode” on page 9-27 for more information).

Figure 7-14. New Display Page



New User Trace Page

This menu choice opens the default application trace page which is automatically pre-configured to show all user events and specific descriptions of the event ID and the first argument of each event.

See “Default Display Page” on page 9-1 for more information.

New Ada Trace Page

This menu choice builds an application trace page 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.

Custom Kernel Page...

Presents the Build Custom Kernel Page dialog (see "Build Custom Kernel Page" on page 7-25) to quickly build a customized kernel page based on choices of nodes, CPUs, and graphs. When loading kernel trace events in NightTrace, default kernel display pages are displayed for each node where trace data originated. These pages show each CPU for each node, as well as a fixed number of graphs and data boxes per CPU.

However, there may be cases where the default display page for kernel data is not desirable:

- on multi-CPU nodes, the vertical height of the default kernel page may be too large
- when shielding a CPU, or running a process with a CPU bias, it may be desirable to see only data for that CPU
- one or more of the default graphs per CPU may not be of interest

See "Kernel Display Pages" on page 12-9 for more information.

Process Specific Page...

Presents the Build Process Specific Kernel Page dialog (see "Build Process Specific Kernel Page Dialog" on page 7-29) to quickly build a customized kernel page that is filtered to display specific processes. The dialog allows you to choose one or more processes from the list of processes represented in the current kernel data set.

Open Existing Page...

This menu choice presents the user with a standard file selection dialog so that they may select a pre-existing configuration page from a previous NightTrace session.

Build Custom Kernel Page

The Build Custom Kernel Page dialog is opened by selecting Custom Kernel Page... from the Pages menu of the NightTrace Main window (see “Custom Kernel Page...” on page 7-24).

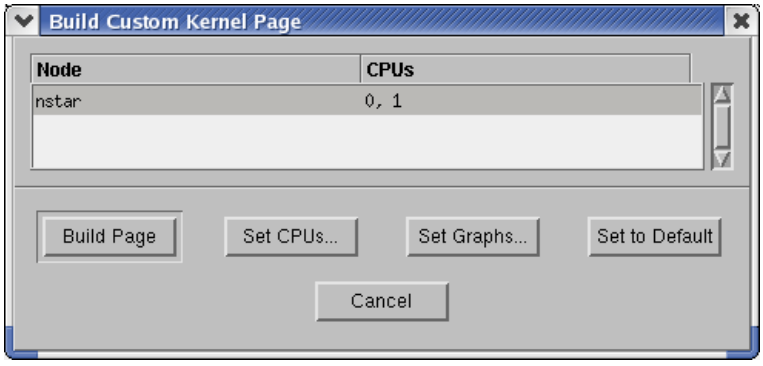


Figure 7-15. Build Custom Kernel Page dialog

Select which nodes you would like included on the customized kernel page from the list.

NOTE

To select multiple items, press the Ctrl key while selecting individual items in the list or hold the Shift key to select a range of items.

Build Page

Creates the customized kernel page based on choices of nodes, CPUs, and graphs.

Set CPUs...

Presents the Select CPUs dialog as shown in Figure 7-16 allowing the user to choose which CPUs from the selected node to display in the display page.

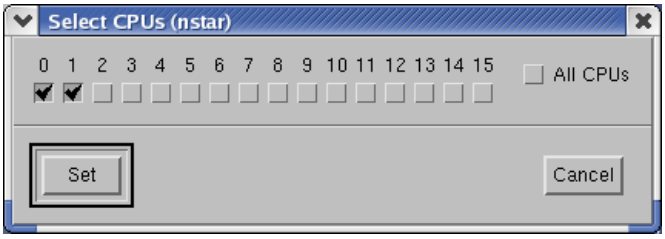


Figure 7-16. Select CPUs dialog

Select the desired CPUs and press the Set CPUs button. Cancel dismisses the dialog without making any changes.

Note for each node, the CPUs that would be displayed in the new kernel display page are shown in the CPUs column of the Build Custom Kernel Page dialog.

NOTE

By default, all CPUs per node are displayed by default in the built kernel display page.

Set Graphs...

Presents the Select Graphs dialog (see "Select Graphs" on page 7-27) allowing the user to choose which graphs to display for each CPU in the display page to be built.

Set to Default

Restores the default settings so that all graphs are displayed for all CPUs.

Cancel

Aborts the building of the customized kernel display page.

Select Graphs

The Select Graphs dialog allows the user to choose which graphs to display for each CPU in the display page to be built.

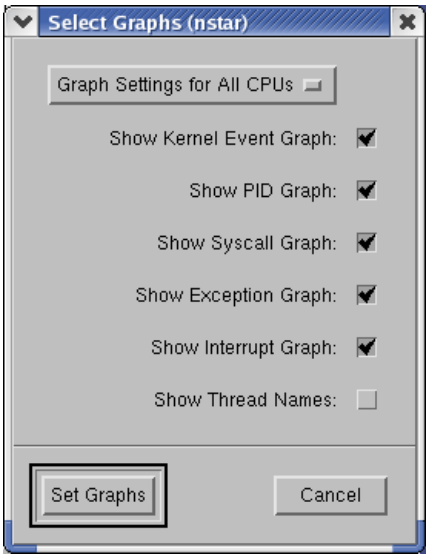


Figure 7-17. Select Graphs dialog

Select the CPU to which these Graph Settings will apply from the drop-down at the top of the dialog and press the Set Graphs button after selecting the desired graphs. Cancel aborts the selection.

Show Kernel Event Graph

When this item is checked, the display page will include a kernel event graph for the CPU(s) selected from the Graph Settings drop-down.

Show PID Graph

When this item is checked, the display page will include a PID graph for the CPU(s) selected from the Graph Settings drop-down.

See “Process Information” on page 12-12 for more information.

Show Syscall Graph

When this item is checked, the display page will include a syscall graph for the CPU(s) selected from the Graph Settings drop-down.

See “System call Information” on page 12-11 for more information.

Show Exception Graph

When this item is checked, the display page will include an exception graph for the CPU(s) selected from the Graph Settings drop-down.

See "Exception Information" on page 12-11 for more information.

Show Interrupt Graph

When this item is checked, the display page will include an interrupt graph for the CPU(s) selected from the Graph Settings drop-down.

See "Interrupt Information" on page 12-10 for more information.

Show Thread Names

When this item is checked, the display page will append the name of the thread associated with an event to the process name in the process label associated with PID graphs. Thread names are only available when mixing user trace data with kernel trace data when the user trace data was generated by a program using the thread-aware version of the NightTrace API library. See "Threads and Logging" on page 2-26 for more information. If a thread name is not available, the thread's system thread ID (see **`gettid(2`**)) is used instead.

Build Process Specific Kernel Page Dialog

The Build Process Specific Kernel Page dialog is opened by selecting Process Specific Page… from the Pages menu of the NightTrace Main window (see “Process Specific Page...” on page 7-24).

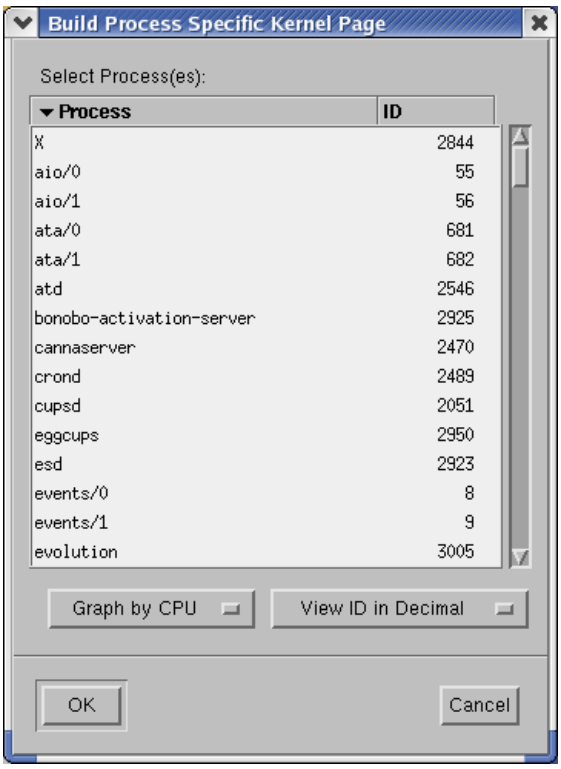


Figure 7-18. Build Process Specific Kernel Page dialog

This dialog will build a kernel display page customized to display information about the processes you select from the list.

The kernel display page will consist of one or more *kernel-information displays*, depending on the Graph By menu selection described below. A kernel-information display consists of 5 rows of labels and graphs which describe interrupts, exceptions, system calls, process information, and kernel events. See “Kernel Display Pages” on page 12-9 for more information on kernel-information displays.

Select the processes you want included on the customized kernel page from the list.

NOTE

To select multiple items, press the Ctrl key while selecting individual items in the list or hold the Shift key to select a range of items.

NOTE

Two columns are presented in the dialog: Process and ID. You can sort the list of processes by clicking on either column header. Clicking on a column header sets the sort criteria to that column and toggles the sort order.

Graph by CPU/Process Menu

This option menu allows you to select how the processes should be displayed on the new kernel page. Selecting Graph by CPU will generate individual kernel-information displays which only display the processes you select; one for each CPU on the system associated with the current kernel data set. Selecting Graph by Process will generate individual kernel-information displays; one for each of the processes you select, regardless of the CPU upon which the process executes.

View ID in Decimal/Hexadecimal

This option menu controls the formatting of the process ID values in the list of processes and process IDs.

OK

Builds a new kernel display page based on the selections you have made in this dialog.

Cancel

Aborts the building of the customized kernel display page.

Profiles

The Profiles menu manipulates the list of profiles shown in the Profiles area of the NightTrace Main window.

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.

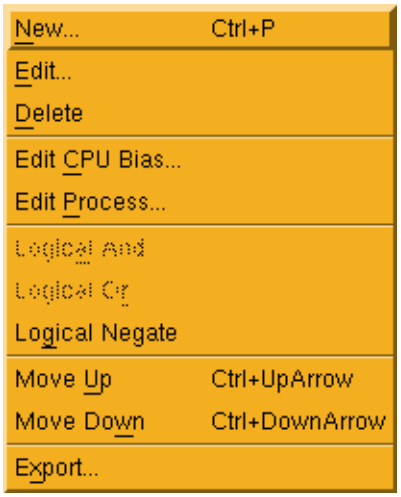


Figure 7-19. Profiles menu

New...

Accelerator: Ctrl+P

This menu choice opens the Profiles dialog so that the user may define a new profile. See Chapter 8, “Profiles” for more information.

Edit...

This menu choice opens the Profiles dialog positioned to the selected profile for editing. See Chapter 8, “Profiles” for more information.

Delete

This menu choice deletes all currently selected profiles.

Edit CPU Bias...

This menu choice allows the user to add or change a CPU condition on the selected profile. This is a convient way to apply a CPU condition to multiple profiles at once.

Edit Process...

This menu choice allows the user to add or change a process condition on the selected profile. This is a convenient way to apply a process condition to multiple profiles at once.

Logical And

This menu choice creates a new profile whose condition is the logical AND of the two currently selected profiles. The new profile is automatically added to the list of profiles in the Profile List area of the NightTrace Main window.

Logical Or

This menu choice creates a new profile whose condition is the logical OR of the two currently selected profiles. The new profile is automatically added to the list of profiles in the Profile List area of the NightTrace Main window.

Logical Negate

This menu choice creates a new profile whose condition is the logical negation of the currently selected profiles. The new profile is automatically added to the list of profiles in the Profile List area of the NightTrace Main window.

Move Up

Accelerator: Ctrl+UpArrow

This menu choice moves the currently selected profile towards the beginning of the list of profiles shown in the Profile List area of the NightTrace Main window.

Move Down

Accelerator: Ctrl+DownArrow

This menu choice moves the currently selected profile towards the end of the list of profiles shown in the Profile List area of the NightTrace Main window.

Export...

This menu choice opens the Export Profiles to Analysis API Source dialog to automatically generate source defining and referencing profiles.

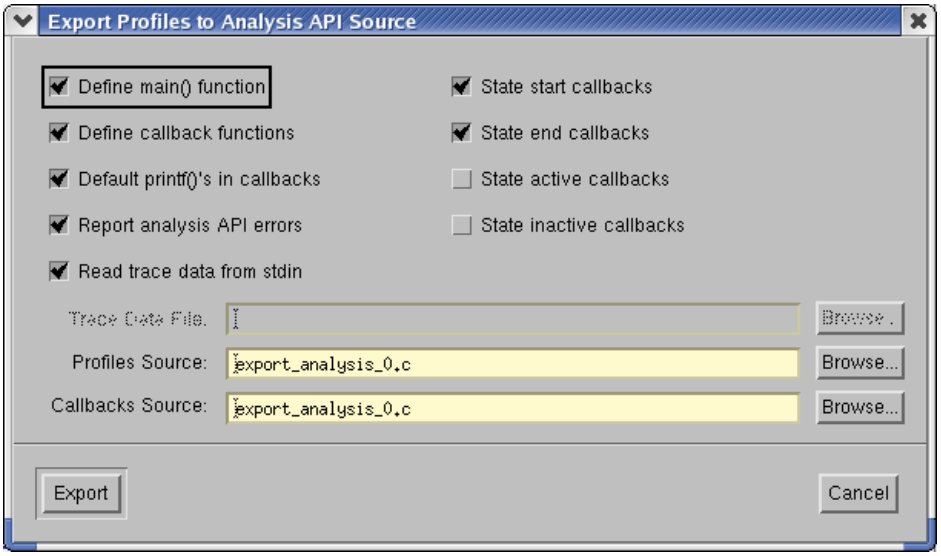


Figure 7-20. Export Profiles dialog

Generate 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.

Generate callback function definitions

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.

Include default printf() output in callbacks

When checked, this option generates source code to print information about instances of the selected profiles in the callback function definitions.

Report errors from API function calls

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.

Callback for state start

When checked, a callback profile is generated and registered with the API for the start event of the selected state profiles.

Callback for state end

When checked, a callback profile is generated and registered with the API for the end event of the selected state profiles.

Callback for state active

When checked, a callback profile is generated and registered with the API for any event that occurs when selected state profiles are active.

Callback for state inactive

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.

Profile 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.

Event

The Event menu aids in traversing events in the data set.

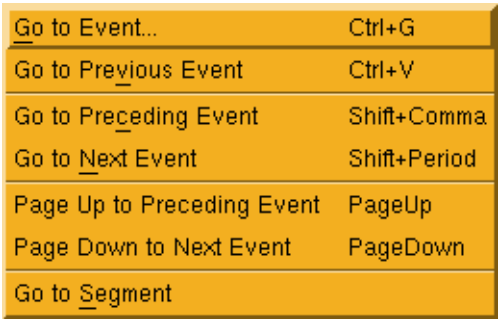


Figure 7-21. Event menu

Go to Event...

Accelerator: Ctrl+G

This menu choice opens a dialog that allows you to enter the event offset or event time of a desired event. Event times are recognized by an “s” suffix, denoting seconds. This causes the current timeline to move to the specified event/time.

Go to Previous Event

Accelerator: Ctrl+V

This menu choice moves the current timeline to the event where the most recent current timeline was previously defined. You can use this menu choice to go back and forth between two events of interest.

Go to Preceding Event

Accelerator: Shift+Comma

This menu choice moves the current timeline to the closest event proceeding the current timeline.

Go to Next Event

Accelerator: Shift+Period

This menu choice moves the current timeline to the closest event after the current timeline.

Page Up to Preceding Event

Accelerator: PageUp

Advance the current timeline backward so the previous set of events are displayed in the Event Text area of the NightTrace Main window. The size of the Event Text area pane determines the number of events to back up.

Page Down to Next Event

Accelerator: PageDown

Advance the current timeline forward so the next set of events are displayed in the Event Text area of the NightTrace Main window. The size of the Event Text area pane determines the number of events to advance.

Go to Segment

Change the current timeline to the first event of the selected trace segment in the Trace Segment area of the NightTrace Main window. This option is useful when you have multiple segments loaded that are spread out over large sections of time.

Edit

The Edit menu provides for customization of string and format tables as well as for managing event annotations.

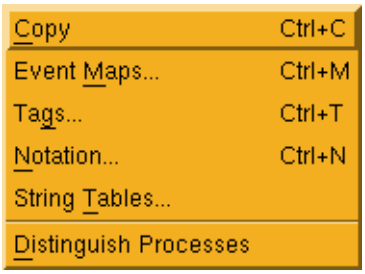


Figure 7-22. Edit menu

Copy

Accelerator: Ctrl+C

This menu choice copies all text associated with the selected events Events area into the inter-application cut and paste buffer.

Event Maps...

Accelerator: Ctrl+M

This menu choice launches the Edit String Table dialog which allows you to change or add textual handles to event ID numbers and control which arguments are printed when event detail is shown. See Section “Edit String Table” on page 9-18 for more information.

Tags...

Accelerator: Ctrl+T

This menu choice launches the Tags dialog which lists all event tags, their time, offset and distance from the current time line, as well as any textual annotations. See Section “Tags” on page 9-14 for more information.

Notation

Accelerator: Ctrl+N

This menu choice tags the currently selected event in the Events area and opens the Edit Tag Notation dialog which allows you to associate text notes with the tag. If the currently selected event already has been tagged, the existing tag is used and no new tag is created. See Section “Tags” on page 9-14 for more information.

String Tables...

This menu choice launches the Edit String Tables dialog which allows you to customize textual information associated with event descriptions and modify thread and process name resolution. See Section “Edit String Tables” on page 9-16 for more information.

Distinguish Processes

This menu choice causes NightTrace to automatically change all process name description to include the system process (or thread) ID as part of the process name, for any process name which refers to more than one thread ID or process ID. For example, if a data set includes events from two processes name **app**, the process name description might be displayed as **app_23983** and **app_23997**.

View

The View menu controls which areas of the NightTrace Main window are visible.

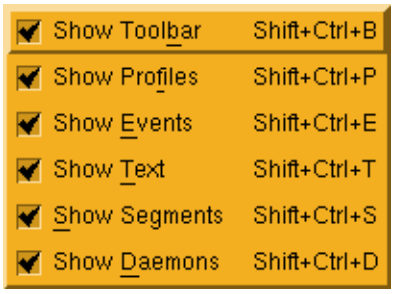


Figure 7-23. View menu

Show Toolbar

This checkbox controls whether the toolbar is displayed.

Show Profiles

This checkbox controls whether the Profiles area is displayed. When NightTrace is initially launched with a new session, no profiles are defined and the Profile area is automatically hidden.

Show Events

This checkbox controls whether the Events area is displayed.

Show Text

This checkbox controls whether the Event Detail area is displayed.

Show Segments

This checkbox controls whether Trace Segment Statistic area is displayed. When launching NightTrace, if no trace data segments yet exist, this area is automatically hidden.

Show Daemons

This checkbox controls whether the Daemon Control area is displayed.

Tools

The Tools menu appears on the NightTrace Main window menu bar (see “NightTrace Main window Menu Bar” on page 7-2).

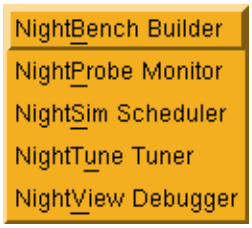


Figure 7-24. Tools menu

NightBench Builder

Opens the NightBench Program Development Environment. NightBench is a set of graphical user interface (GUI) tools for developing software with the Concurrent MAXAda™ compiler.

See also:

- *NightBench User’s Guide* (0890480)

NightProbe Monitor

Opens the NightProbe Data Monitoring application. NightProbe is a real-time graphical tool for monitoring, recording, and altering program data within one or more executing programs without significant intrusion. NightProbe can be used in a development environment as a tool for debugging, or in a production environment to create a “control panel” for program input and output.

See also:

- *NightProbe User’s Guide* (0898480)

NightSim Scheduler

Opens the NightSim Application Scheduler. NightSim is a tool for scheduling and monitoring real-time applications which require predictable, repetitive process execution. With NightSim, application builders can control and dynamically adjust the periodic execution of multiple coordinated processes, their priorities, and their CPU assignments.

See also:

- *NightSim User’s Guide* (0890480)

NightTune Tuner

Opens the NightTune system analysis and tuning application. NightTune is a real-time graphical tool for monitoring system, process, and thread activity. NightTune provides dynamic tuning of process and thread scheduling attributes as well as CPU shielding and CPU interrupt affinity.

See also:

- *NightTune User's Guide* (0898515)

NightView Debugger

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

See also:

- *NightView User's Guide* (0898395)

Help

The Help menu appears on the NightTrace Main window menu bar (see “NightTrace Main window Menu Bar” on page 7-2).

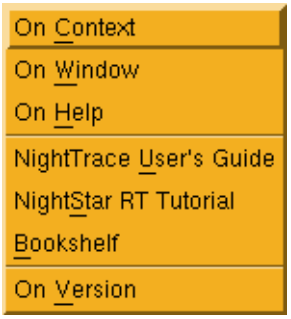


Figure 7-25. Help menu

On Context

Gives context-sensitive help on the various menu options, dialogs, or other parts of the user interface.

Help for a particular item is obtained by first choosing the On Context 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).

In addition, context-sensitive help may be obtained for the currently highlighted option by pressing the F1 key. The HyperHelp viewer will display the appropriate topic.

On Window

Displays the help topic for the current window.

On Help

Displays this section of the on-line help manual.

NightTrace User’s Guide

Opens the online *NightTrace User’s Guide*.

NightStar RT Tutorial

Opens the online *NightStar RT Tutorial*.

Bookshelf

Opens a HyperHelp window that lists all of the Concurrent online publications currently available on the local system.

On Version

Displays version and copyright information for the NightTrace product.

NightTrace Tool Bar

The NightTrace tool bar provides icons for commonly used actions.

Figure 7-26. NightTrace Main Tool Bar

Save the current NightTrace session

Create a new daemon

Search backward

Search forward

Summarize

Open profile dialog to create a new profile

Move selected profile(s) down

Move selected profile(s) up

Go to event

Move timeline to previous location

Move timeline to preceeding event

Move timeline to next event

Move timeline; page up in the Event area

Move timeline; page down in the Event area

Tag selected event and open annotation dialog

Open Tags dialog

Open the Edit Event Map dialog

Open the Edit String Tables dialog

The tool bar may be hidden by clearing the Show Toolbar checkbox in the View menu of the NightTrace Main window. Alternatively, the keyboard sequence Shift+Ctrl+B toggles the visibility of the tool bar.

Profile Area

The profile area lists all currently defined profiles. Profiles are conditions and states that have been defined or used in search and summary operations.

The profile area is a resizable pane which contains a scrollable list of profiles with 7 columns of information.

Type

The type column indicates whether the profile defines a state or just a condition.

Name

The name column indicates the name of the profile. Names are automatically assigned when profiles are created and can be changed using the Profiles dialog.

Search

Indicates the profile which will be used if a search operation is executed.

Status

Indicates whether to profile conditions are currently true or false for the event at the or immediately to the left of the current timeline.

Count

Indicates the count of matches of the profiles from the beginning of the data set up to the current timeline.

Last Offset

Indicates the last offset which matched the conditions specified in the profile.

Start Offset

For state profiles, indicate the closest start offset of the associated state at or immediately to the left of the current timeline.

Double-clicking a row in the profiles list causes that profile to be selected and launches the Profiles dialog for editing, search, or summary.

The Profiles area can be hidden by clearing the Show Profiles checkbox from the View menu of the NightTrace Main window. Alternatively, the keyboard sequence Shift+Ctrl+P toggles the visibility of the Profiles area.

Event Area

The event area lists individual trace events.

The event area is a resizable pane which contains a scrollable list of events with 7 columns of information.

Offset

The offset of the event is shown.

Event

The event name or number is shown.

CPU

For kernel events, the CPU on which the event was logged is shown. For non-kernel events, the CPU number is not available.

Process

The name of the process or process ID of the event is shown.

Thread

The thread name of the event or system thread ID of the event is shown. For kernel events, thread names are only available if user trace data sets are added and the application logging the events uses the *thread-aware* version of the NightTrace Logging API. See “Threads and Logging” on page 2-26 for more information.

Time (sec)

Indicates the time within the data set of the event in seconds.

Tag

Indicates the name of the tag, if any, associated with the event.

The event in the middle of the list which is marked with ---> on the left and highlighted in a salmon background is the event at the current timeline.

Using the UpArrow and DownArrow keys causes the current timeline to advance one event, either backward or forward, respectively.

Using the PageUp and PageDown keys causes the current timeline to advance to the previous or next *page* of events, respectively. A *page* in this case being defined by the number of events visible in the events list.

Right-clicking a row in the events list causes a new tag to be created and associated with the event.

Selecting a row in the list causes the Event Detail area to display event details related to the newly selected event.

The Events area can be hidden by clearing the Show Events checkbox from the View menu of the NightTrace Main window. Alternatively, the keyboard sequence Shift+Ctrl+E toggles the visibility of the Events area.

Event Detail Area

The event detail area lists additional information about the selected event in the Event area.

The event detail area is a resizable pane which contains a scrollable list of event information which includes the following lines:

Offset

The offset of the event is shown.

Detail

For kernel events, a descriptive explanation is included.

Args

The value of any arguments associated with the event are displayed.

Tag

For tagged events, the tag name is displayed.

Notation

For tagged events with annotations, the annotation is displayed.

The Event Detail area can be hidden by clearing the Show Text checkbox from the View menu of the NightTrace Main window. Alternatively, the keyboard sequence Shift+Ctrl+T toggles the visibility of the Event Detail area.

Trace Segment Statistic Area

The trace segment area lists individual statistics about individual data segments in the data set.

The trace segment area is a resizable pane which contains a scrollable list of segments with 7 columns of information.

Type

This column contains an indicator of the type of data in the segment, either **K** for kernel data and **U** for user data.

Trace Segment

This column contains the name of the data segment. If the data segment came from a file, the name of the file is used. If it came from a streaming daemon, then the daemon handle from the Daemon Control area is used.

Count

This column contains the number of events in the data segment.

Lost

This column contains the number of events that were lost. Event loss can occur when the daemon cannot copy events quickly enough from the memory buffers to the output device. See "Preventing Trace Event Loss" on page 5-1 for more information.

Duration

This column displays the timespan of events in the trace data segment in units of seconds.

Target

This column displays the system from which the trace data was generated.

Unsaved

This column contains an indicator when the trace data segment has not yet been saved. This only occurs when streaming live data directly into NightTrace.

Double-clicking an entry in the list launches the Trace Segment Header Information dialog, which displays detailed information about the trace data segment.

The Trace Segment Statistic area can be hidden by clearing the Show Segments checkbox from the View menu of the NightTrace Main window. Alternatively, the keyboard sequence Shift+Ctrl+S toggles the visibility of the Trace Segment Statistic area.

Daemon Control Area

The daemon control area displays information about the daemons defined in the current session and allows you to control their execution.

NightTrace allows users to manage user and kernel NightTrace daemons using *daemon definitions* which are saved as part of the *session* in the session configuration file (see "Session Configuration Files" on page 6-24). These definitions include daemon collection modes and settings, daemon priorities and CPU bindings, and data output formats, as well as the trace event types that are logged by that particular daemon.

Individual daemons within a session may or may not be related to each other in any meaningful way. One might use a session simply to hold several daemon definitions that are commonly used, but not necessarily all at the same time.

Users can manage multiple daemons simultaneously on multiple target systems from a central location and may start, stop, pause, and resume execution of any of the daemons under its management. The user may also view statistics as trace data is being gathered as well as dynamically enable and disable events while a particular daemon is executing.

In addition to sending trace output to a file for later analysis, NightTrace also offers a *streaming* output method. When streaming, trace output is sent directly to the NightTrace display buffer for immediate analysis even while additional trace data is being collected.

Double-clicking or selecting and pressing Enter on an entry in the Daemon Control Area brings up the Daemon Definition Dialog for the daemon associated with that entry (see "Daemon Definition Dialog" on page 7-57).

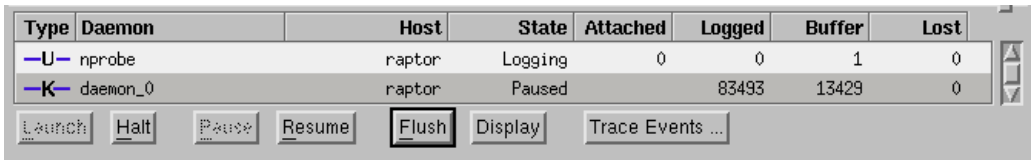


Figure 7-27. Daemon Control Area

Type

Indicates what type of trace events the daemon is logging.

U — indicates that the associated daemon is logging user trace events

K — indicates that the associated daemon is logging kernel trace events

The type of trace event that the daemon is logging is configured by selecting either the Kernel or the User Application radio button in the Trace section on the General page of the Daemon Definition dialog (see "General" on page 7-62).

Daemon

The name of the daemon as configured in the Name field on the General page of the Daemon Definition dialog (see "Name" on page 7-62).

NOTE

The Daemon is merely a label to aid the user in identifying specific daemons with a session. It has no external meaning and is unrelated to the NightTrace API.

Target

The name of the system on which the associated daemon is running.

The target system is specified in the Target System field on the General page of the Daemon Definition dialog (see "Target System" on page 7-63).

State

The state of the daemon.

Logging	indicates the daemon is currently capturing events
Halted	indicates the daemon is not executing
Paused	indicates the daemon is started but is not capturing events While paused, attempts to log events from user applications or via the operating system kernel are discarded. Note that these are not considered lost events (see "Lost" on page 7-52).
Pausing	indicates the daemon is going from a Logging state to a Paused state
Resuming	indicates the daemon is going from a Paused state to a Logging state
Launching	indicates the daemon is going from a Halted state to a Logging state
Halting	indicates the daemon is going from a Paused or Logging state to a Halted state

Attached

The number of user application threads or processes that are associated with the daemon.

Logged

The number of trace events that have been written to the stream or written to the file by the associated daemon.

Buffer

The number of trace events currently held in the trace buffers that has not yet been copied to the trace file or stream.

These events can be explicitly flushed from the buffers by pressing the Flush button.

Lost

Lost events occur when the daemon cannot keep up with the rate at which events are being added to the buffer. See "Preventing Trace Event Loss" on page 5-1 for more information.

NOTE

Events that are discarded when a daemon is Paused (see "State" on page 7-51) are not included in the Lost count.

Also, events that are discarded when the daemon is in Buffer Wrap mode (see "Buffer Wrap" on page 7-65) (i.e. older events being discarded in favor of new ones) are not included in the Lost count.

The area located at the bottom of the Daemon Control Area contains a number of buttons which control the daemons currently selected in the Daemon Control Area.

Launch

Accelerator: Ctrl+L

Starts execution of the daemon(s) currently selected in the Daemon Control Area.

NOTE

Starting a daemon does not imply that the daemon begins to collect events.

Launch operations are time consuming and involve possibly connecting to a target system, user authentication, etc. Once the daemon is launched, it is more efficient to utilize the Pause and Resume operations which require less time and resources.

Halt

Accelerator: Ctrl+H

Stops execution of the daemon(s) currently selected in the Daemon Control Area.

The connection to the target system is terminated by this operation. Once the daemon is launched, it may be more efficient to utilize the Pause and Resume operations.

Pause

Pauses the execution of the daemon(s) currently selected in the Daemon Control Area.

NOTE

When a daemon is paused, incoming trace events are discarded without notice.

Resume

Accelerator: Ctrl+R

Resumes execution of the daemon(s) currently selected in the Daemon Control Area. Once resumed, incoming events are placed into the daemon buffer for subsequent processing by the daemon.

Flush

Flushes trace events from the buffers associated with the daemon(s) currently selected in the Daemon Control Area to either the NightTrace display buffer (see “Stream” on page 7-64) or to the output file (see “Output File” on page 7-65).

Display

When data from the selected daemon(s) is being streamed to the NightTrace display buffer (as specified by the setting of the Stream checkbox on the General page of the Daemon Definition dialog (see “General” on page 7-62)), pressing this button causes a flush of the data currently in the trace buffer to the NightTrace display buffer. If no display pages currently exist, a default display page will be created when this button is pressed.

NOTE

The user must scroll the NightTrace display or zoom out in order to see the most up-to-date data.

When data from the selected daemon(s) is written to output files, pressing this button causes the data in the output file to be displayed in the NightTrace display.

Trace Events...

Presents the Enable/Disable Trace Events dialog (see "Enable / Disable Trace Events" on page 7-55) allowing the user to dynamically enable or disable selected trace event types while a particular daemon is running. A currently executing daemon must be selected from the Daemon Control Area.

Enable / Disable Trace Events

The Enable/Disable Trace Events dialog allows the user to dynamically enable or disable selected trace event types while a particular daemon is running. This dialog is opened by selecting a currently executing daemon from the Daemon Control Area and pressing the Trace Events... button in the Daemon Control Area of the NightTrace Main window (see “Daemon Control Area” on page 7-50).

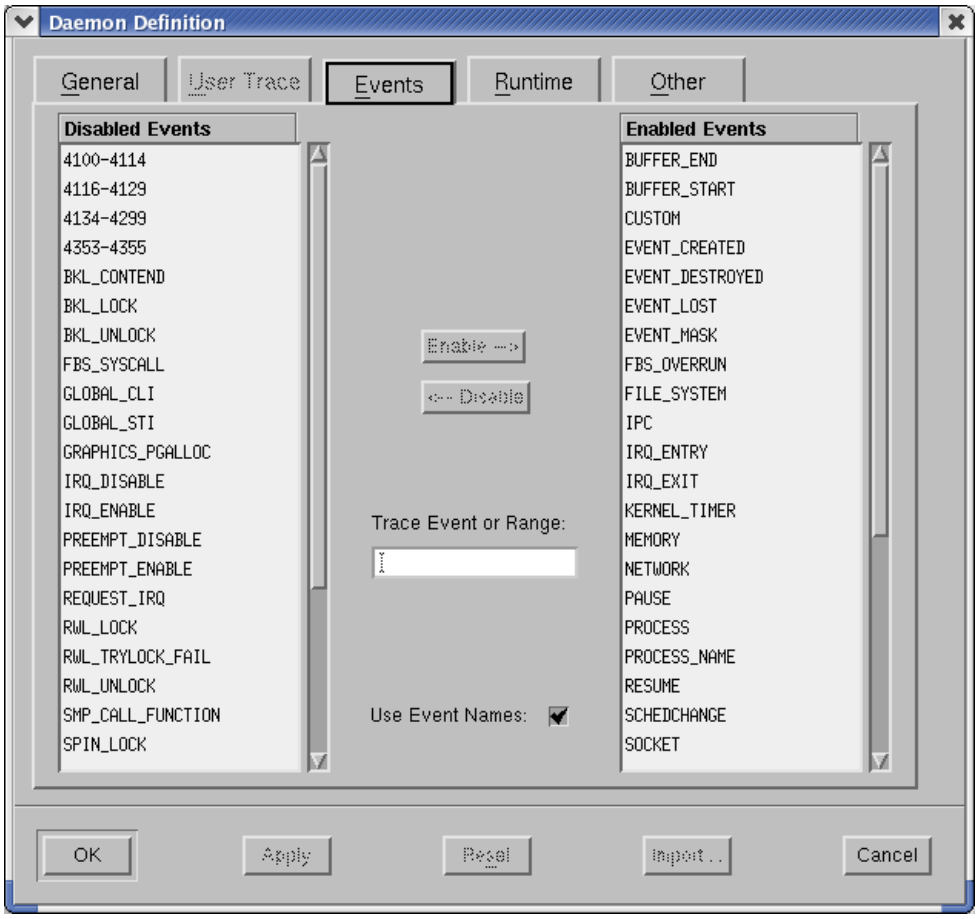


Figure 7-28. Enable / Disable Trace Events dialog

Disabled Events

This is a list of user trace or kernel trace event types that are disabled.

Disabled events are not logged to daemon buffers and therefore are not included in event trace outputs.

Enabled Events

This is a list of user trace or kernel trace event types that are enabled.

Enabled events are allowed to be placed into daemon buffers and are subsequently transferred to the output device (see "Trace Events Output" on page 7-64).

Enable -->

Moves the selected items from the Disabled Events list or the Trace Event or Range field to the Enabled Events list.

<-- Disable

Moves the selected items from the Enabled Events list or the Trace Event or Range field to the Disabled Events list.

Trace Event or Range

Allows the user to enter a particular trace event type (or range of trace event types) and subsequently Enable --> or Disable --> it.

The user may use the event name associated with the event type (e.g. SYSCALL_RESUME) or the numerical value of the trace event type (e.g. 4131).

The user may also enter a range of values either using the event names or their numerical values (e.g. IRQ_ENTRY-IRQ_EXIT or 4305-4306).

Use Event Names

Allows the user to view the event names of the trace event types in the Disabled Events and Enabled Events lists instead of their numerical values.

For user trace events, the user may load user-defined event map files which associate meaningful names with the user trace event ID numbers (see "Event Map Files" on page 6-11).

Daemon Definition Dialog

The Daemon Definition dialog allows the user to create and modify the various aspects of a daemon configuration.

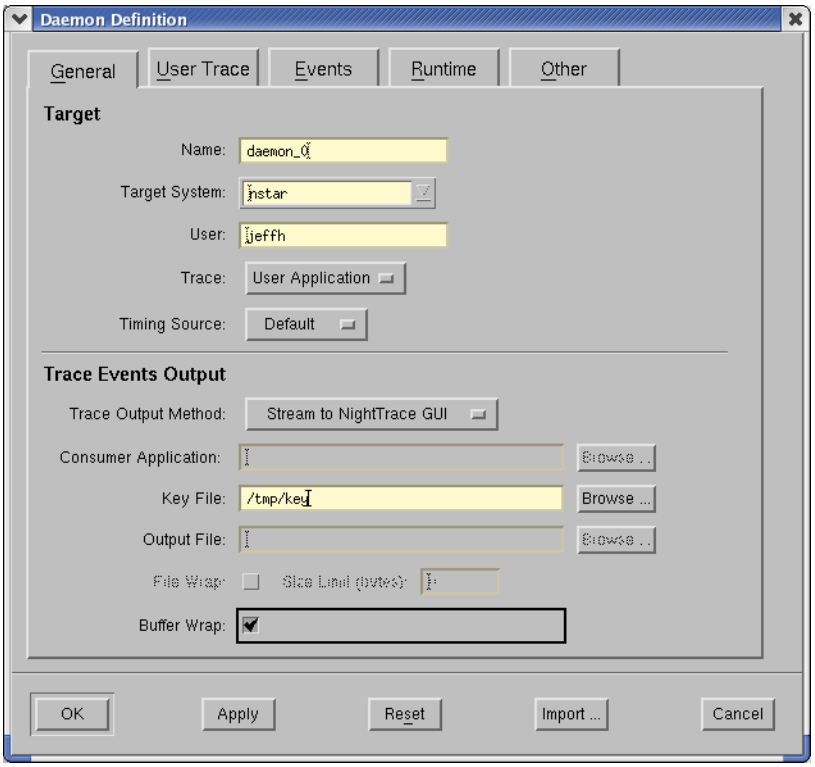


Figure 7-29. Daemon Definition dialog

The Daemon Definition dialog is divided into a number of pages that contain specific information about the current configuration. These pages are:

- General

  This page contains information such as the name of the daemon configuration, the target system on which the daemon will run, the user's login on that system, and settings specifying whether kernel or user application tracing will be performed. Items related to trace events output such as the names of output and key files and settings such as whether or not *streaming* will be performed by this daemon are found on this page as well.

  See "General" on page 7-62 for more detailed information.

- User Trace

  This page contains settings for user trace daemons such as locking policies associated with the daemon, shared memory permissions, and the duration of the timestamp heartbeat, as well as specifications for the size and flush threshold of the user event buffer.

  See "User Trace" on page 7-67 for more detailed information.

- Events

  This page allows the user to specify which events may be logged while tracing.

  See "Events" on page 7-70 for more detailed information.

- Runtime

  This page allows the user to specify the scheduling policy, priority and CPU bias for the daemon.

  See "Runtime" on page 7-72 for more detailed information.

- Other

  This page allows the user to specify advanced settings with respect to the transfer of trace data from the daemon to the NightTrace display buffer.

  See "Other" on page 7-74 for more detailed information.

The following buttons appear at the bottom of the Daemon Definition dialog and have the specified meaning:

OK

This button applies changes made and closes the Daemon Definition dialog.

Apply

This button applies changes made but leaves the Daemon Definition dialog open.

Reset

This button restores the values of all items to the previously-applied values and leaves the Daemon Definition dialog open.

Import...

Presents the Import Daemon Definition dialog (see "Import Daemon Definition" on page 7-60) allowing the user to define daemon attributes based on a user application running on a remote system.

Cancel

This button restores the values of all items to the previously-applied values and closes the Daemon Definition dialog.

Import Daemon Definition

This dialog allows the user to define daemon attributes based on a running user application containing NightTrace API calls. The Import Daemon Definition dialog is presented following user authentication (see "Login" on page 7-18 and "Enter Password" on page 7-18).

The user may select an application, running on the specified target system, from which they wish to import trace-related attributes.

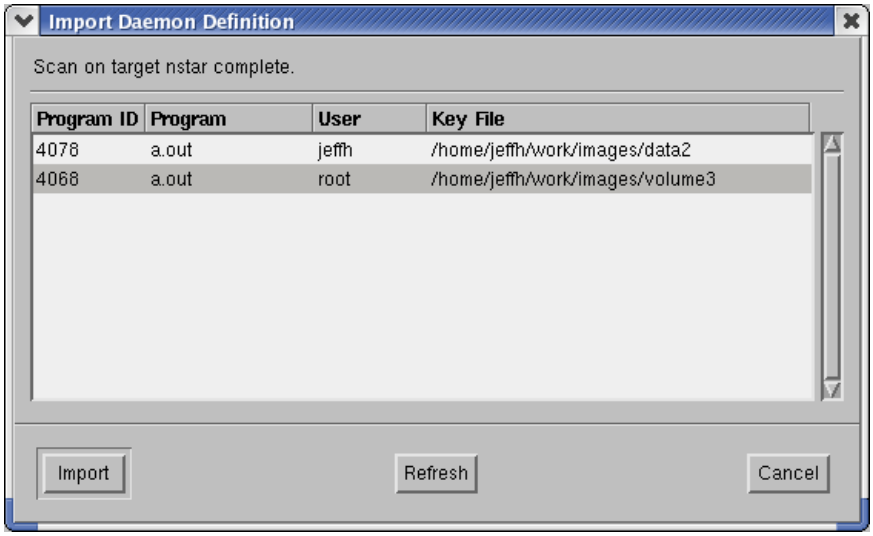


Figure 7-30. Import Daemon Definition dialog

Program ID

The process ID (PID) of the Program on the remote system.

Program

The name of the user application containing `trace_` calls on the remote system.

User

The user who invoked the Program on the remote system.

Key File

The filename which is used to calculate the shared memory segment identifier associated with the logging of user trace events. See "Key File" on page 7-64 for more information.

The following buttons appear at the bottom of the Import Daemon Definition dialog and have the specified meaning:

OK

Imports daemon attributes into the current daemon definition from the user application selected in the list.

Refresh

Queries the specified target system for user applications making trace-related calls.

Cancel

This button closes the Import Daemon Definition dialog without importing any daemon attributes from any of the listed applications.

Help

Brings up online help for this dialog.

General

The General page of the Daemon Definition dialog (see "Daemon Definition Dialog" on page 7-57) contains information such as the name of the daemon configuration, the target system on which the daemon will run, the user's login on that system, and settings specifying whether kernel or user application tracing will be performed. Items related to trace events output such as the names of output and key files and settings such as whether or not *streaming* will be performed by this daemon are found on this page as well.

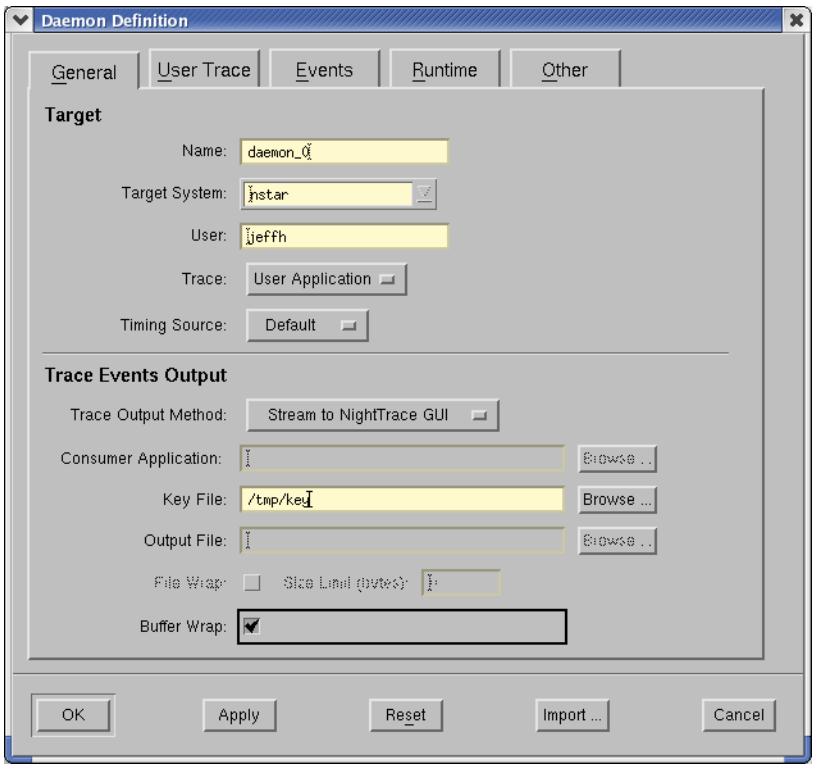


Figure 7-31. Daemon Definition dialog - General

Target

Name

The name for this daemon definition.

This field is automatically populated with the name daemon_*x* where *x* is a number, starting at 0, which increments with each new daemon definition.

The Name is merely a label to aid the user in identifying specific daemons with a session. It has no external meaning and is unrelated to the NightTrace API. The user may change this to a name of their choosing.

Target System

The system on which this trace daemon will run.

User

The name of the user on the specified Target System responsible for running this daemon.

Trace

Indicates what type of trace events this daemon will be logging.

Kernel

Indicates that the daemon is logging kernel trace events.

Kernel events are automatically generated by the operating system kernel when a kernel daemon is initiated if the operating system kernel was built with tracing support.

For systems running RedHawk Linux, see the *Concurrent Real-Time Linux - RT User Guide* (0898004) for more detailed information.

User Application

Indicates that the daemon is logging user trace events.

User trace events are generated by:

- user applications that use the NightTrace API
- the NightProbe tool (see the description of the To NightTrace menu item in the chapter titled “Using the Data Recording Window” in the *NightProbe User’s Guide*.
- the Nightview tool (see the description of the TracePoint menu item) in the *NightView User’s Guide*.

Timing Source

By default, an architecture-specific clock is used to timestamp trace events. On iHawk systems, the Time Stamp Counter is used.

NightTrace can also specify the Real-Time Clock and Interrupt Module (RCIM) as a timestamp source (see “Timestamps” on page 1-2 for more information). This is most useful when concurrent traces running on multiple systems are desired. Using the RCIM as a timing device allows NightTrace to present the user with a synchronized view of concurrent activities on those systems.

Default

On iHawk systems, the Intel Time Stamp Counter is used.

RCIM Tick

Specifies that the Real-Time Clock and Interrupt Module (RCIM) tick clock will be used to timestamp trace events.

NOTE

Use of this option requires that an RCIM board is installed and configured on the target system.

Trace Events Output

Stream

When checked, this specifies that *streaming* is in effect so that the output trace events will go directly to the NightTrace display buffer. Otherwise, the output will be written to the Output File (see below).

Key File

Specifies a filename which is used to calculate the shared memory segment identifier associated with the logging of user trace events. The daemon and the NightTrace API use the **`ftok(3)`** service to map the specified filename to a shared memory identifier as used by **`shmat(2)`**.

NOTE

When the output method is NOT *streaming* (see Stream above), the Key File defines the name of the Output File where trace events are written (see "Output File" on page 7-65).

The Key File is relative to the target system. It does not necessarily need to be accessible from the *host system* (the system where the NightTrace GUI is running); however, that can be convenient for subsequent analysis via NightTrace.

Furthermore, the Key File does not have to pre-exist. If a user application has not already created it via a NightTrace API call, the daemon will create the file if it does not exist.

Browse...

Brings up a standard file selection dialog so that the user may navigate to the desired location of the Key File.

In order to browse, the Target System (see "Target System" on page 7-63) must be operational. The file selection dialog invoked by that button shows files relative to the Target System.

Output File

The name of the file to which trace events are written.

The Output File is relative to the target system. It does not necessarily need to be accessible from the *host system* (the system where the NightTrace GUI is running); however, that can be convenient for subsequent analysis via NightTrace.

NOTE

When the output method is NOT *streaming* (see Stream above), the Key File (see "Key File" on page 7-64) defines the name of the Output File.

Browse...

Brings up a standard file selection dialog so that the user may navigate to the desired location of the Output File.

In order to browse, the Target System (see "Target System" on page 7-63) must be operational. The file selection dialog invoked by that button shows files relative to the Target System.

File Wrap

When checked, allows the user to specify the Maximum File Size for the Key File/Output File.

Maximum File Size

The maximum number of bytes for the Key File/Output File.

When the Maximum File Size is reached, subsequent events will overwrite the oldest events. NightTrace automatically detects this and presents events in chronological order, from oldest to newest. Events that are discarded due to File Wrap are not considered "lost events" (see "Lost" on page 7-52) in statistics provided by the NightTrace.

NOTE

For a daemon capturing kernel trace events, the file wrap sizes that the user specifies are rounded up to a multiple of kernel buffer sizes.

Buffer Wrap

When this is checked, the daemon will overwrite the least recently recorded events in the trace buffer when it reaches its maximum size.

For user trace events, the size and number of shared memory buffers are specified in the Number of Buffers and Buffer Size fields on the User Trace page of the Daemon Definition dialog (see "User Trace" on page 7-67).

For kernel trace events, the size and number of kernel buffers are defined in the Trace Buffer Size and Number of Buffers fields on the Other page of the Daemon Definition dialog (see "Other" on page 7-74).

User Trace

The User Trace page of the Daemon Definition dialog (see “Daemon Definition Dialog” on page 7-57) contains settings for locking policies associated with the daemon and the corresponding user applications using the NightTrace API, shared memory permissions, as well as specifications of the size and number of user event buffers.

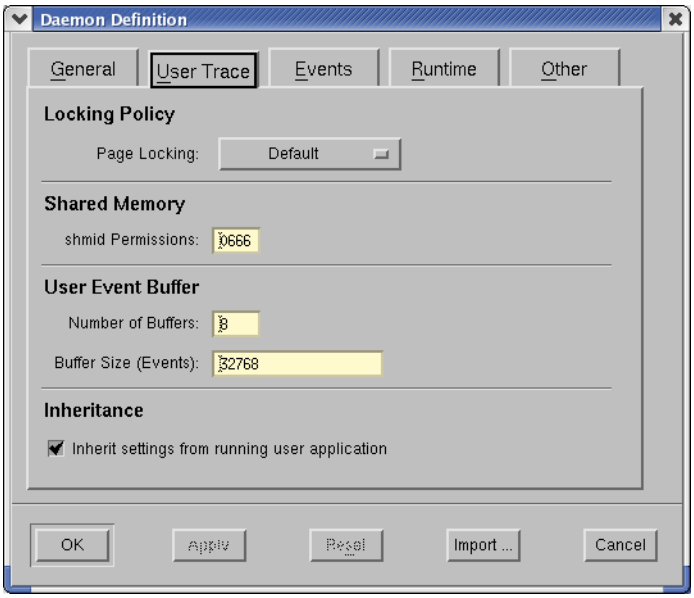


Figure 7-32. Daemon Definition dialog - User Trace

Locking Policy

Page Locking

When the Lock Critical Pages option is selected, the daemon and user applications associated with this daemon lock down the required pages and unlock them when the NightTrace API is terminated. Selecting the Default option will allow pages to be locked if the user application has already specified this option and is already executing, otherwise no page locking will occur. Selecting No Locking will ensure that pages are not locked even if the user application has specified page locking and has already begun executing -- in this case, the daemon will fail to launch with an appropriate diagnostic.

Shared Memory

The daemon and the user applications communicate with each other through shared memory. The shared memory segment identifier is calculated from the Key File setting (see “Key File” on page 7-64).

The shared memory segment is automatically destroyed after the last user application and/or the daemon exits normally (if the daemon or user applications are aborted, the shared memory segment will remain; it will be reinitialized on subsequent use).

shmid Permissions

If the daemon is initiated before any user applications, then the shared memory segment will be created with the specified permissions.

User Event Buffer

Number of Buffers

The number of buffers used to hold events generated by the application.

The specified value should be a power of 2. If it is not, the value is automatically rounded up to the next power of 2.

Buffer Size

The size of each buffer used to hold event generated by the application in units of *raw events*. API calls which log just an event ID consume the space required by one raw event. Other API calls which provide additional parameters with the event ID require additional space.

The specified value should be a power of 2. If it is not, the value is automatically rounded up to the next power of 2.

Inheritance

When the daemon starts up, certain settings can be inherited from a running user application that has set up a trace definition.

The NightTrace API `trace_begin()` call allows the user to define the following settings in a user application:

- those values listed under the Page Locking categories on this page
- the Number of Buffers and Buffer Size also found on this page
- the setting for the Timing Source which appears on the General page of the Daemon Definition dialog (see "General" on page 7-62)

See "trace_begin" on page 2-6 for more information on this API.

Inherit settings from running user application

When this is checked, trace settings defined by a running user application are silently preferred if those definitions differ from those made in NightTrace.

If not checked, trace settings defined by user applications must match those in the current daemon definition.

See above for details on which trace settings may be inherited.

Events

The Events page of the Daemon Definition dialog (see "Daemon Definition Dialog" on page 7-57) allows the user to specify which trace event types will be handled by the daemon.

The user may also change this list dynamically while the daemon is executing by pressing the Trace Events... button in the Daemon Control Area of the NightTrace Main window (see "Daemon Control Area" on page 7-50) to bring up the Enable/Disable Trace Events dialog (see "Enable / Disable Trace Events" on page 7-55).

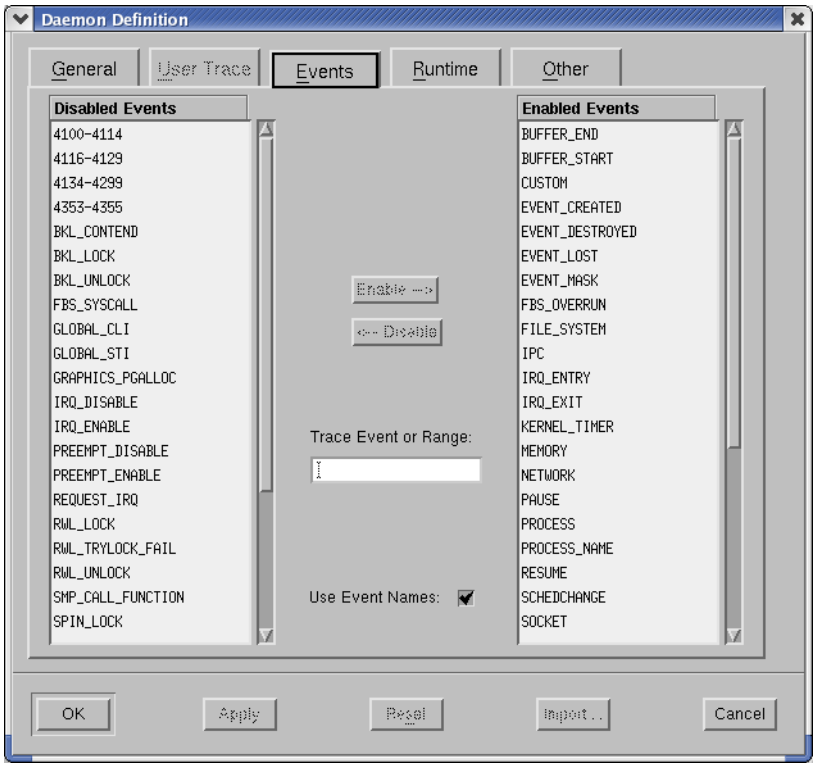


Figure 7-33. Daemon Definition dialog - Events

Disabled Events

This is a list of user trace or kernel trace event types that are disabled.

Disabled events are not logged to daemon buffers and therefore are not included in event trace outputs.

Enabled Events

This is a list of user trace or kernel trace event types that are enabled.

Enabled events are allowed to be placed into daemon buffers and are subsequently transferred to the output device (see “Trace Events Output” on page 7-64).

Enable -->

Moves the selected items from the Disabled Events list or Trace Event or Range field to the Enabled Events list.

<-- Disable

Moves the selected items from the Enabled Events list or Trace Event or Range field to the Disabled Events list.

Trace Event or Range

Allows the user to enter a particular trace event type (or range of trace event types) and subsequently Enable --> or Disable --> it.

The user may use the event name associated with the event type (e.g. SYSCALL_RESUME) or the numerical value of the trace event type (e.g. 4131).

The user may also enter a range of values either using the event names or their numerical values (e.g. IRQ_ENTRY-IRQ_EXIT or 4305-4306).

Use Event Names

Allows the user to view the event names of the trace event types in the Disabled Events and Enabled Events lists instead of their numerical values.

For user trace events, the user may load user-defined event map files which associate meaningful names with the user trace event ID numbers (see “Event Map Files” on page 6-11).

Runtime

The Runtime page of the Daemon Definition dialog (see "Daemon Definition Dialog" on page 7-57) allows the user to specify the scheduling policy, CPU bias, and memory binding policies for the daemon.

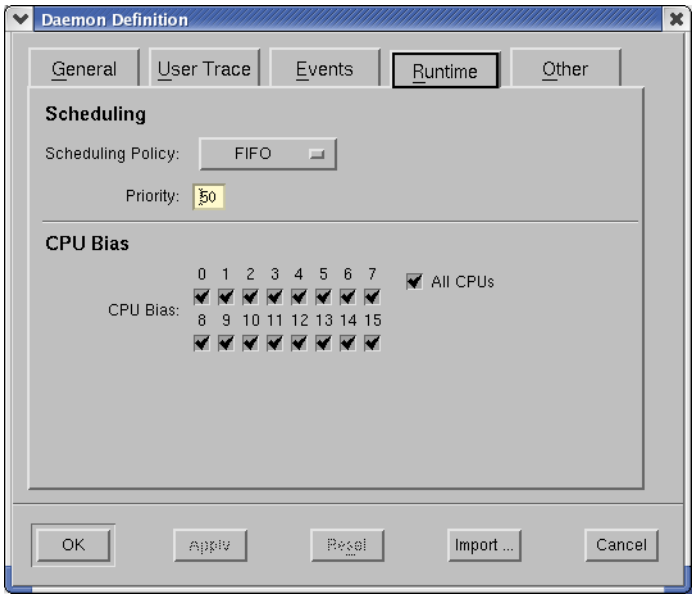


Figure 7-34. Daemon Definition dialog - Runtime

Scheduling

Scheduling Policy

POSIX defines three types of policies that control the way a process is scheduled by the operating system. They are SCHED_FIFO (FIFO), SCHED_RR (Round Robin), and SCHED_OTHER (Time-Sharing). Each of these scheduling policies is associated with one of the System V scheduler classes. See the *RedHawk Linux User's Guide* (0898004) for more detailed information regarding these policies and their associated classes.

FIFO

The FIFO (first–in–first–out) policy (SCHED_FIFO) is associated with the fixed-priority class in which critical processes 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.

Time-Sharing

The Time-Sharing policy (SCHED_OTHER) is associated with the time-sharing class, changing priorities dynamically and assigning time slices of different lengths to processes in order to provide good response time to interactive processes and good throughput to CPU-bound processes.

Priority

The Priority is relative to the selected Scheduling Policy (see "Scheduling Policy" on page 7-72) 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

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.

Other

The Other page of the Daemon Definition dialog (see "Daemon Definition Dialog" on page 7-57) allows the user to specify advanced settings with respect to the transfer of trace data from the daemon to the NightTrace display buffer.

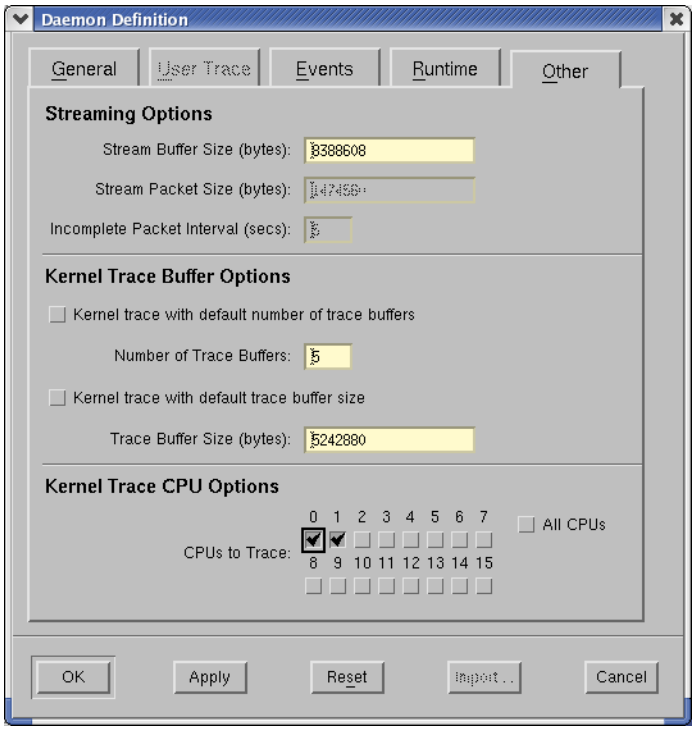


Figure 7-35. Daemon Definition dialog - Other

Streaming Options

Stream Buffer Size

The number of bytes for the buffer that the NightTrace uses to hold data from the daemon before sending it to the NightTrace display buffer.

NOTE

This is an internal buffer. You should not need to adjust the size of this buffer unless NightTrace finds that it cannot transfer data quickly enough between the daemon and the NightTrace analyzer. In such a circumstance, the daemon is forced into a Paused state (see "State" on page 7-51).

Stream Packet Size

The amount of data (in bytes) sent from the daemon to the NightTrace analyzer for individual I/O transfers. Different network configurations may have different optimal packet sizes.

Incomplete Packet Interval

This setting is intended for applications that have very low event rates. The user may not want to wait for a full packet (specified by the Stream Packet Size) to fill before the data is sent to the analyzer. If a packet cannot be filled in this amount of time, the available trace data is sent anyway.

NOTE

The user can always hit the Flush button (see "Flush" on page 7-53) which causes all data in the trace buffer to be immediately transmitted across the stream.

Kernel Trace Buffer Options

The kernel modules collect data into one or more trace buffers as events are logged. The trace daemon started by the server (**`ntraceserv`**) either writes events from these buffers to a file or stream.

Increasing the following settings should help avoid data loss.

Kernel trace with default number of trace buffers

On Linux, if the default is used, the number of trace buffers used by the kernel modules to collect data defers to the server (**`ntraceserv`**) which starts the daemon.

Number of Trace Buffers

The desired number of trace buffers used by the kernel modules to collect data.

Kernel trace with default trace buffer size

On Linux, if the default is used, the default trace buffer size defers to the server (**`ntraceserv`**) which starts the daemon.

Trace Buffer Size

The desired size of the trace buffers used by the kernel modules to collect data.

Kernel Trace CPU Options

By default, kernel tracing logs events that occur on all CPUs. You can restrict the CPUs where tracing occurs by selecting individual CPU checkboxes.

8
Profiles

The Profiles dialog provides a centralized point of control to select, create, manage, and utilize profiles.

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.

Profiles include any condition or state you use within a NightTrace session, including those used in search and summary operations.

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
- Display page graph objects

The Profiles dialog consists of the following areas:

- The profile menu bar
- The profile tool bar
- The profile text area
- The profile definition area
- The profile action control area

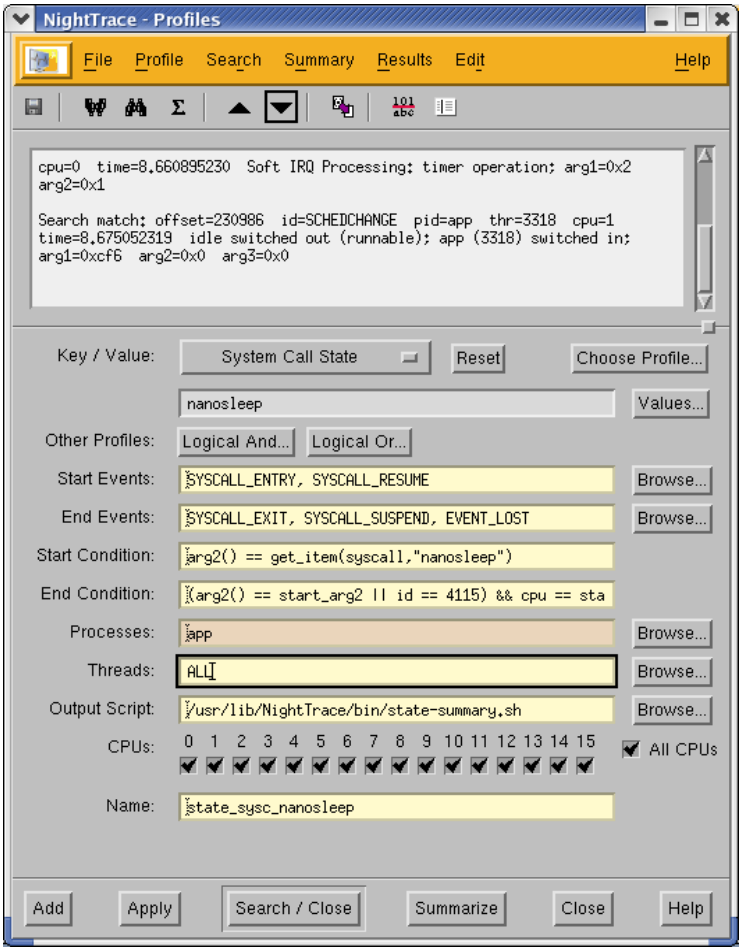


Figure 8-1. Profiles Dialog

Profile Menu Bar

The Profiles dialog menu bar provides access to the following menus:

- File
- Profile
- Search
- Summary
- Results
- Edit
- Help

Each menu is described in the sections that follow.

File Menu

The File menu allows you to save the current session, raise the NightTrace Main window and close the Profiles dialog.

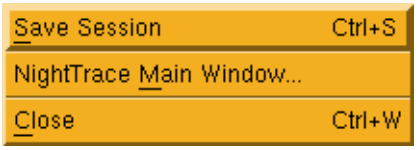


Figure 8-2. File menu

Save Session...

Accelerator: Ctrl+S

Save Session saves the current session to a session configuration file (see "Session Configuration Files" on page 6-24 for a complete description of the contents of a session).

Save Session allows for quickly saving a session. The user is not prompted for the filenames where the session, trace data, or display pages are to be saved. These are automatically saved in appropriately named files in the current working directory.

If the current session has not been saved to a file in the past, the session is automatically saved to a new session configuration file. The new filename appears in the 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 (see "Discard Events..." on page 9-12). 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.

NightTrace Main Window

NightTrace Main Window opens the NightTrace Main window and brings it to the front of the screen.

Close

Accelerator: Ctrl+W

Close closes the Profiles dialog.

If modifications have been made in the Profiles dialog and have not yet been saved via an Add, Apply, Search/Close or Summarize action, the changes are discarded.

Profile Menu

The Profile menu allows you to select from previously-defined profiles and export profiles in the form of NightProbe Analysis API source code.

Figure 8-3. Profile menu

Choose Profile...

Accelerator: Ctrl+P

Choose Profile allows you to select from a list of all previously defined profiles from the Choose Profile dialog as shown below:

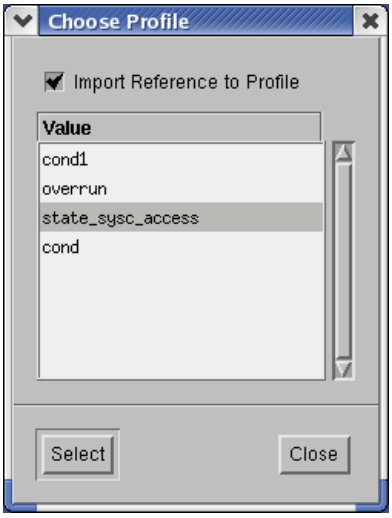


Figure 8-4. Chose Profile dialog

Selecting a profile from the list will cause the Profiles dialog to refer to the selected profile. If the Import Reference to Profile checkbox is checked, the Profiles dialog will be set to use an expression that refers to the selected profile. Otherwise, the entire profile is copied into the Profiles dialog and it becomes the current profile. Use the former technique to add additional constraints on an existing profile without disassociating from that profile. Thus subsequent changes to the selected profile will still be reflected in the new profile. Use the latter technique to create a copy of the selected profile or to modify that profile.

View Next Profile

Accelerator: UpArrow

This menu option populates the Profiles dialog with the previous profile in the circular list of profiles.

View Previous Profile

Accelerator: DownArrow

This menu option populates the Profiles dialog with the next profile in the circular list of profiles.

Export...

This menu choice opens the Export Profiles to Analysis API Source dialog to automatically generate source code defining and referencing the profile currently displayed in the Profiles dialog. See “Export...” on page 7-32 for more information.

Search Menu

The Search menu allows you to search forward or backward for the current profile and set search options.

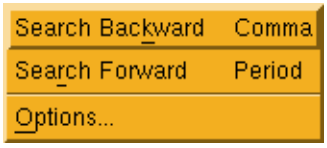


Figure 8-5. Search menu

Search Backward

Accelerator: Comma

Search backward for the profile as defined in the Profiles dialog.

Search Forward

Accelerator: Period

Search forward for the profile as defined in the Profiles dialog.

Options...

Options launches the Search Options dialog which allows you to define the search scope and control search wrap options. See "Options..." on page 7-9 for more information.

Summary Menu

The Summary menu allows you to execute a summary for the current profile and set summary options.

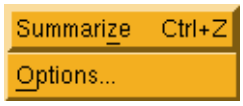


Figure 8-6. Summary menu

Summarize

Accelerator: Ctrl+Z

Execute a summary for the profile as defined in the Profiles dialog.

Options

Options launches the Summary Options dialog which allows you to define the summary scope and control summary display options. See “Options...” on page 7-11 for more information.

Results Menu

The Results menu manipulates the results text area.

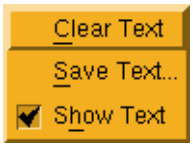


Figure 8-7. Results menu

Clear Text

Clears the summary and search text area.

Save Text

Saves the search and summary text from the results text area to a file. A standard file selection dialog is displayed.

Show Text

The Show Text checkbox controls whether the search and summary text area is visible in the Profiles dialog.

Edit Menu

The Edit menu allows you to launch support dialogs which control how event information is displayed.

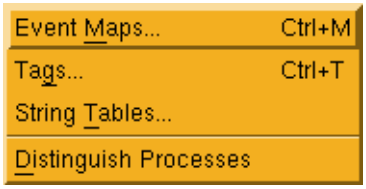


Figure 8-8. Edit menu

Event Maps...

Accelerator: Ctrl+M

This menu item launches the Edit String Table dialog which allows you to change or add textual handles to event ID numbers and control which arguments are printed when event detail is shown. See Section "Edit String Table" on page 9-18 for more information.

Tags...

Accelerator: Ctrl+T

This menu item launches the Tags dialog which lists all event tags, their time, offset and distance from the current time line, as well as any textual annotations. See Section "Tags" on page 9-14 for more information.

String Tables...

This menu item launches the Edit String Tables dialog which allows you to customize textual information associated with event descriptions. String tables are also useful in forming expressions used in profiles and display graph objects. See Section "Edit String Tables" on page 9-16 for more information.

Distinguish Processes

This menu item causes NightTrace to automatically change all process name description to include the system process (or thread) ID as part of the process name, for any process name which refers to more than one thread ID or process ID. For example, if a data set includes events from two processes name **app**, the process name description might be displayed as **app_23983** and **app_23997**.

Help Menu

The Help menu allows you to obtain help on any portion of the Profiles dialog or on other topics. The menu is common between the NightTrace Main window, the Profiles dialog, and Display page windows. See "Help" on page 7-41 for more information.

Profile Tool Bar

The Profiles tool bar provides icons for commonly used actions.

Figure 8-9. Profiles Tool Bar


Save the current NightTrace session


Search backward for the current profile


Search forward for the current profile


Summarize the current profile


Populate the dialog with the previous profile in the circular list of profiles


Populate the dialog with the next profile in the circular list of profiles


Open Tags dialog


Open the Edit Event Map dialog


Open the Edit String Tables dialog

Profile Text Area

The profile text area shows the results of search and summary operations. The area may be scrolled to view previously executed searches and summaries.

Text in the area may be saved to a file using the Save Text… option from the Results menu.

Text in the area may be cleared using the Clear Text option from the Results menu.

The profile text area itself may be hidden or shown using the Show Text checkbox in the Results menu.

Profile Definition Area

This area 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.

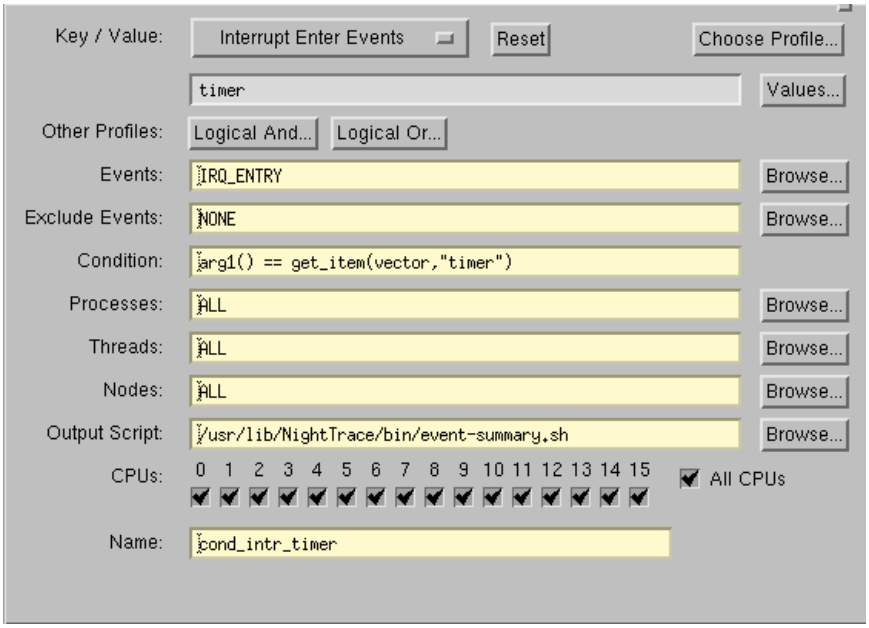


Figure 8-10. Profile Definition Area

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 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.

These options desensitized if kernel trace data is not loaded.

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 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.

These options desensitized if kernel trace data is not loaded.

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 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.

These options desensitized if kernel trace data is not loaded.

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.

You can tag events with labels and annotations using the Tag icon on the tool bar, the Tags… option from the Edit dialog, as well as other actions in the NightTrace main window and in Display pages (see page 7-47 for information).

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 dialog 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 Reference to Profile checkbox in the Choose Profile… list, the dialog 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.

Other Profiles

This area allows you to create a new profile with additional constraints associated with a previously-defined condition.

Pressing Logical And… or Logical Or… launches a list of known profiles and imports the profile you select by reference into the dialog, combining it with the current profile via a boolean AND or OR operation, respectively.

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 11-1 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 2-26 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 area. Output from *stderr* is not captured.

Summary data is passed to the specified script via environment variables. See "Summary Script Environment Variables" on page 8-20 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.

Action Control Area

The Action Control Area allows you to create new profiles, apply changes to existing profiles, and execute search and summary actions.

Add

The Add button creates a new profile based on the conditions in the Profiles dialog. If another profile with the same name already exists, the new profile's name 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 Profiles dialog. If the profile did not previously exist, it adds the profile.

Search/Close

The Search/Close button executes a forward search for the selected profile and closes the Profiles dialog.

The scope of the search, the actions to be taken upon a match, and wrapping options can be modified using the Options... option of the Search menu item (see "Options..." on page 8-6 for more information).

When the Profiles dialog is launched from the Search... menu choice of the Search menu (or via Ctrl+F), the default focus for the action control buttons is on the Search/Close button. This allows you to quickly define your search criteria and execute the search. For example, the following keyboard and mouse sequences might be used:

1. Press Ctrl+F to launch the Profiles dialog for searching
2. Left-Click the Key/Value option list
3. Select System Call All Events
4. Select a system call from the popup dialog
5. Press Enter to close the pop-up dialog
6. Press Enter to execute the search and close the Profiles dialog

Forward and backward searches on the current profile can be executed directly from the NightTrace Main window and from Display Page window, without launching the Profiles dialog, by pressing the Period and Comma key, respectively. Additionally, forward and backward search icons appear in the tool bars of those windows.

NOTE

To execute a backward search from within the Profiles dialog, Add or Apply the profile condition, if changed, and then press the Backward Search icon on the tool bar or use the Search Backward option from the Search menu.

Summarize

Accelerator: Ctrl+Z

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.

The scope, action, and results options can be modified using the Options… option of the Summary menu.

See "Summarizing Statistical Information" on page 8-19 for more information.

Close

Accelerator: Esc

The Close button closes the Profiles dialog, discarding any profile changes that have not been applied or added.

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 Options

The scope, timeline actions, graphical actions, and results options can be modified using the Options… option of the Summary menu.

See "Options..." on page 7-11 for more information.

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 8-16 for more information on specifying user-defined scripts.

Summary Script Environment Variables

The following summary environment variables are passed to summary scripts

Table 8-1. 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.

Table 8-1. Summary Script Environment Variables

Variable	Meaning
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.
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.

9
Display Pages

A *display page* lets you view trace event data by allowing you to:

- create and configure display objects to graphically depict your trace session (see Chapter 10 “Display Objects”)
- define profiles of conditions and states (see Chapter 8 “Profiles”) to aid in the analysis of trace data
- search for certain trace events based on specific critiera (see “Search/Close” on page 8-17)
- summarize data into statistical information regarding particular trace events and states (see “Summarizing Statistical Information” on page 8-19)

Default Display Page

The default display page contains a number of preconfigured display objects (see Chapter 10 “Display Objects”) that allow you to analyze your trace data with minimal effort. If this page does not exactly meet your needs, you can modify it according to your specifications. NightTrace brings up this page in *view mode* (see “Switch between view and edit mode” on page 9-27 for more information).

A default display page contains a Grid Label (see “Grid Label” on page 10-4) and a State Graph (see “State Graph” on page 10-7) for each registered thread logging trace events in your trace event file(s). Each State Graph is configured to display only those events logged by a particular registered thread; the associated Grid Label identifies that thread. An additional State Graph is also created which is configured to display all user events from all threads combined. If the number of threads is so large that their associated State Graphs will not all fit on the grid, then NightTrace stops creating state graphs.

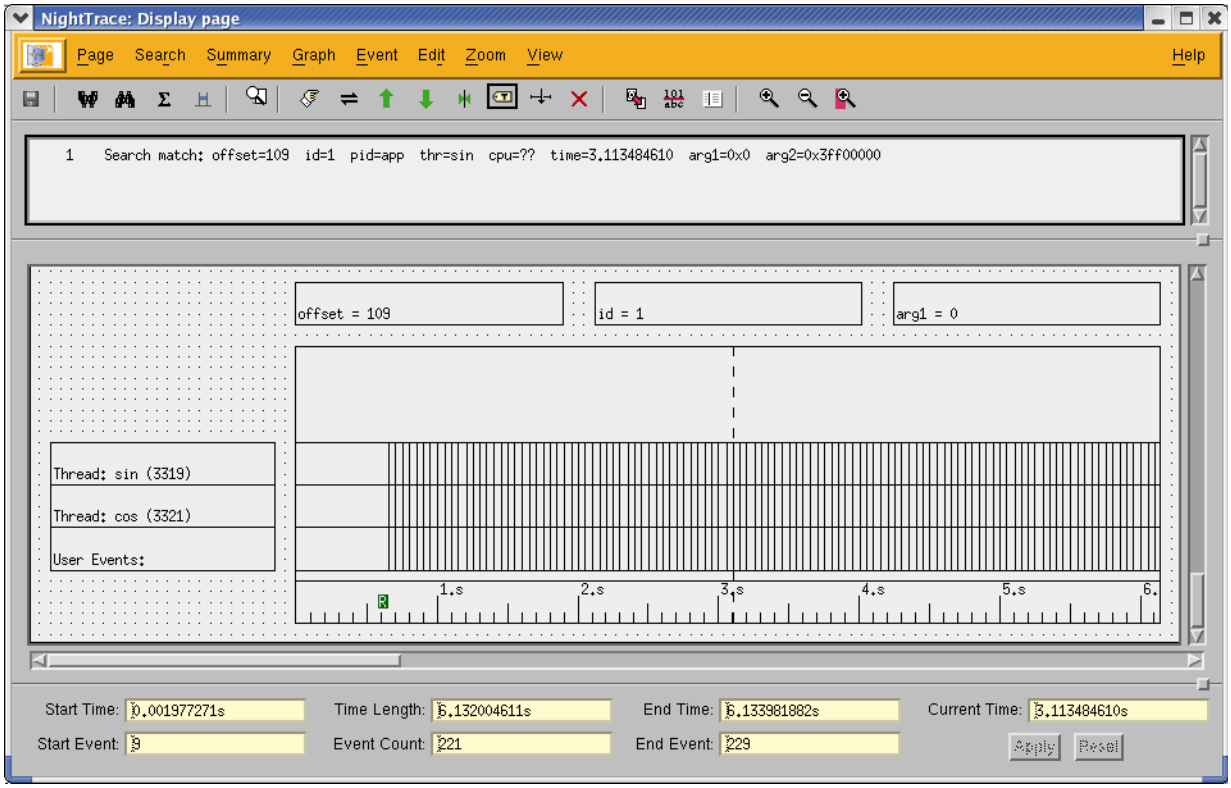
NOTE:

To distinguish between individual threads, threads must be registered. See “Threads and Logging” on page 2-26 for more information.

In addition, Data Boxes (see “Data Box” on page 10-5) appear at the top of the default display page containing information related to the *current trace event*. This information includes the *offset*, *trace event ID*, and first *trace event argument* logged by that particular trace event.

Figure 9-1 shows a default display page for two threads, `sin` and `cos`, logging trace events. The information in the Data Boxes at the top of the grid relate to the last trace event on or before the current time line. A State Graph has been created showing the trace events logged by the thread `jane`; another has been created showing those logged by `tarzan`. A third State Graph appears below the others displaying the trace events logged by both threads.

Figure 9-1. A Default Display Page



A display page consists of the following components:

- Menu Bar (see "Menu Bar" on page 9-3)
- Tool Bar (see "Display Page Tool Bar" on page 9-26)
- Message Display Area (see "Message Display Area" on page 9-29)
- Grid (see "Grid" on page 9-29)
- Interval Scroll Bar (see "Interval Scroll Bar" on page 9-31)
- Interval Control Area (see "Interval Control Area" on page 9-32)

Menu Bar

The menu bar on all display pages provides access to the following menus:

- Page (see "Page" on page 9-3)
- Search (see "Search" on page 9-5)
- Summary (see "Summary" on page 9-6)
- Graph (see "Graph" on page 9-7)
- Event (see "Event" on page 9-9)
- Edit (see "Edit" on page 9-13)
- Zoom (see "Zoom" on page 9-24)
- View (see "View" on page 9-25)
- Help (see "Help" on page 9-26)

Page

The Page menu allows you save the current NightTrace session, save changes to the current display page, and raise and close windows.

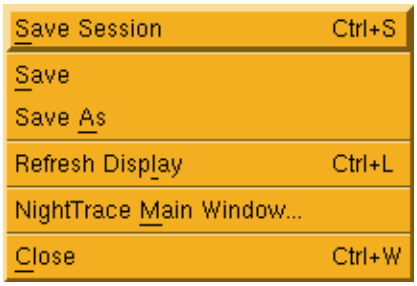


Figure 9-2. Display Page - Page Menu

Save Session

Accellerator: Ctrl+S

Save Session saves the current session to a session configuration file (see "Session Configuration Files" on page 6-24 for a complete description of the contents of a session).

Save Session allows for quickly saving a session. The user is not prompted for the filenames where the session, trace data, or display pages are to be saved. These

are automatically saved in appropriately named files in the current working directory.

If the current session has not been saved to a file in the past, the session is automatically saved to a new session configuration file. The new filename appears in the Trace Segment Statistics area above the Daemon Control Area in the NightTrace Main window (see "Trace Segment Statistic Area" on page 7-48).

If the current session was loaded from or previously saved to a session configuration file, the session is saved to that file.

Trace data that has been *touched* is saved by Save Session. Touched trace data includes trace data modified by discarding events (see "Discard Events..." on page 9-12).

If the trace data was loaded from a previously saved trace data segment file, the data is saved to that file.

If the trace data has never been saved to a trace data segment file, the data is automatically saved to a newly created trace data segment file

Display pages are saved by Save Session. These display pages include those pages created by the Custom Kernel Page menu item under the Pages menu of the NightTrace Main window (see "Custom Kernel Page..." on page 7-24) as well as any modified pages.

If the display page was loaded from a previously saved display page file, the page is saved to that file.

If the display page has never been saved to a display page file, the page is automatically saved to a newly created display page file

Save

Saves the current display page configuration (see "Configuration Files" on page 6-14) to the external file specified with the Save As… menu item. Any changes you have made since the last save operation will be saved to that file; this menu item is disabled (desensitized) if no changes have been made.

This menu item is also disabled if this is a new display page; in this case, use the Save As …. menu item to specify a filename.

Save As...

Presents a file selection dialog to specify a filename to which the current display page configuration will be saved (see "Configuration Files" on page 6-14).

Refresh

Accellerator: Ctrl+L

This menu choice refreshes the display page. Display pages may become stale after entering Edit mode and making configuration changes.

NightTrace Main Window...

Opens the NightTrace Main Window if not currently opened; otherwise, brings the NightTrace Main Window to the foreground.

See Chapter 7 “The NightTrace Main Window” for more information.

Close

Ends the current editing/viewing session, resets all field and radio button settings, and clears the message display area. If you have unsaved changes, a warning dialog box appears, asking if you want to save your changes.

Search

The Search menu allows you to search forward or backward for the current profile and set search options.

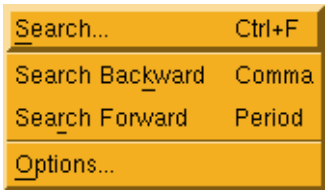


Figure 9-3. Display Page - Search Menu

Search...

Accelerator: Ctrl+F

Displays the Profiles dialog allowing the user to define the search criteria and to execute a search. See “Profiles” on page 8-1 for more information.

Search Backward

Accelerator: Comma

Search backward for the profile as defined in the Profiles dialog.

Search Forward

Accelerator: Period

Search forward for the profile as defined in the Profiles dialog.

Options...

Options launches the Search Options dialog which allows you to define the search domain and control search wrap options. See "Options..." on page 7-9 for more information.

Summary

The Summary menu allows you to execute a summary for the current profile and set summary options.

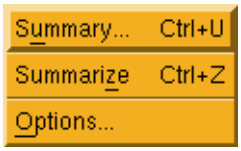


Figure 9-4. Display Page - Summary Menu

Summary...

Accelerator: Ctrl+U

Opens the Profiles dialog (see "Profiles" on page 8-1) allowing the user to select a profile to summarize or define a new profile to summarize.

Summarize

Accelerator: Ctrl+Z

Execute a summary for the profile as defined in the Profiles dialog.

Options

Options launches the Summary Options dialog which allows you to define the summary domain and control summary display options. See "Options..." on page 7-11 for more information.

Graph

The Graph menu manipulates how events are displayed, allowing you to create and modify labels, state, data, and event graphs, and data boxes.

Figure 9-5. Display Page - Graph Menu

Select All

Selects every display object on the grid. This is useful when you want to perform some operation on every display object on the grid (for example, moving or deleting every display object).

NOTE

This operation is enabled only when the display page is in *edit mode* (see “Switch between view and edit mode” on page 9-27).

Deselect All

Deselects every selected display object on the grid.

NOTE

This operation is enabled only when the display page is in *edit mode* (see “Switch between view and edit mode” on page 9-27).

Configure...

Opens the configuration dialog(s) for the selected display object(s).

NOTE

Double-clicking on a particular display object will bring up the configuration dialog for that display object.

See "Configuring Display Objects" on page 10-15 for details.

This operation is enabled only when the display page is in *edit mode* (see "Switch between view and edit mode" on page 9-27).

Copy

Accelerator: Ctrl+C

Copy the selected display object allowing the user to subsequently paste a copy of the display object on the Grid using the Paste menu item.

Only one display object may be copied at a time. In addition, Column display objects (see "Column" on page 10-6) cannot be copied.

NOTE

This operation is enabled only when the display page is in *edit mode* (see "Switch between view and edit mode" on page 9-27).

Paste

Accelerator: Ctrl+V

When this menu item is selected, the mouse pointer becomes a crosshair allowing the user to position and size the display object previously copied using the Copy menu item.

See "Creating Display Objects" on page 10-12 for more information.

NOTE

This operation is enabled only when the display page is in *edit mode* (see "Switch between view and edit mode" on page 9-27).

Delete

Deletes the selected display object(s).

This operation is enabled only when the display page is in *edit mode* (see “Switch between view and edit mode” on page 9-27).

Grid Label

Allows the user to add a Grid Label to the current display page.

See “Grid Label” on page 10-4 for more information.

Data Box

Allows the user to add a Data Box to the current display page.

See “Data Box” on page 10-5 for more information.

Column

Allows the user to add a Column to the current display page.

See “Column” on page 10-6 for more information.

Event Graph

Allows the user to add a Event Graph to the current display page.

See “Event Graph” on page 10-6 for more information.

State Graph

Allows the user to add a State Graph to the current display page.

See “State Graph” on page 10-7 for more information.

Data Graph

Allows the user to add a Data Graph to the current display page.

See “Data Graph” on page 10-8 for more information.

Ruler

Allows the user to add a Ruler to the current display page.

See “Ruler” on page 10-10 for more information.

Event

The Event menu allows you to traverse the trace data set, change the current timeline, mark and tag events, and discard uninteresting events.

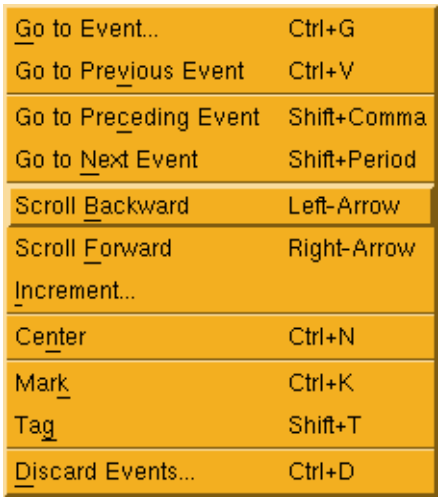


Figure 9-6. Display Page - Event Menu

Go To Event...

Acceleartor: Ctrl+G

This menu choice allows you to enter an event offset or time and causes the current timeline to be moved to the specified event or time. The value entered is considered to be an event offset if it is an integer value. A floating point value is interpretted as an event time. If the value entered refers to an event or time earlier than the first event or later than the last event, the current timeline is moved to the beginning or end of the data set, respsectively.

Go To Previous Event

Accelerator: Ctrl+V

This menu choice moves the current timeline to the event that was previously the assocated with the current timeline. This allows you to flip the timeline back and forth between two events of interest.

Go To Preceeding Event

Accelerator: Shift+Comma

This menu choice moves the current timeline to the immediately preceeding event.

Go To Next Event

Accelerator: Shift+Period

This menu choice moves the current timeline to the next event.

Scroll Forward

Accelerator: right-arrow

Scrolls the interval forward Increment seconds or Increment percent of the current display interval allowing you to examine different intervals in your trace session (see “Increment...” on page 9-11).

See “Interval Scroll Bar” on page 9-31 for related information.

Scroll Backward

Accelerator: left-arrow

Scrolls the interval backward Increment seconds or Increment percent of the current display interval allowing you to examine different intervals in your trace session (see “Increment...” on page 9-11).

See “Interval Scroll Bar” on page 9-31 for related information.

Increment...

Controls how much the current interval scrolls (and the slider moves) when you:

- click on an arrowhead of the interval scroll bar (see “Interval Scroll Bar” on page 9-31)
- click between an arrowhead and the slider on the interval scroll bar
- select either the Scroll Forward or Scroll Backward menu item from the Event menu of any display page (see “Event” on page 9-9)
- use the < or > accelerator keys to scroll forward or backward (Note that it is not necessary to press the Shift key when using these accelerators.)

This field may contain either a percentage or an absolute amount of time in seconds. The default is 25%.

A valid change keeps percentages greater than 0% and less than or equal to 100% and absolute numbers greater than 0 microseconds and less than or equal to the end time of the trace session. If you set Increment to the word `default` or a space, NightTrace resets Increment to the default value.

If Increment is less than 100% when you click on an interval scroll bar arrowhead, you see part of the previous interval in this interval; if Increment is equal to 100%, you see a completely new interval.

Center

Accelerator: Ctrl+N

This menu choice adjusts the interval start and end values such that the current timeline is centered in the interval. The current timeline does not move in time, just in position on your screen.

Mark

Accelerator: Ctrl+M

This menu choice sets the Mark to the value of the current timeline. There is only one Mark which is used to defined a *region*. Alternatively, you can tag as many events you wish with Tags and add annotations to them. The Mark appears as triangle on the Ruler. A region is defined as the events between the Mark and the current timeline. You can see the time between the cursor and the Mark by using the Ctrl+Middle keyboard/mouse sequence.

Tag

Accelerator: Shift+T

This menu choice creates a new tag at the location of the current timeline. The tag is automatically given a unique tag number and appears on the Ruler as a black rectangle with the tag number displayed in white. Double-clicking on the tag marker opens the Edit Tag Notation dialog for that tag.

Alternatively, you use Right+Click the mouse on an individual event in the display and the Create New Tag dialog is launched which allows you to name the tag and associate it with the event. Tag names can be changed using the Tags dialog.

Discard Events...

Accelerator: Ctrl+D

This menu choice launches the Discard Events... dialog.

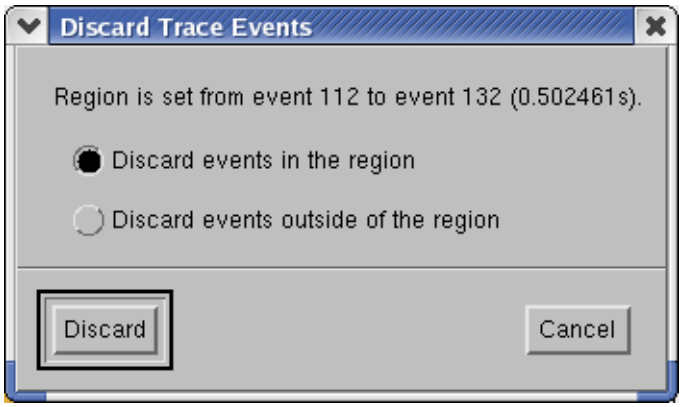


Figure 9-7. Discard Events dialog

The offset and timespan of the current *region* (the events included between the Mark and the current timeline) are described in the dialog.

You can discard the events within the region or all events outside of the region.

Discarded events are immediately removed from the loaded data set, however, if the data set was originally loaded from a file, the file will not be changed unless you save the session.

It is often convenient to reduce the data set size to just events of interest using Discard Events... and then save a copy of the session using the Save Session Copy menu choice from the NightTrace menu in the NightTrace Main window. This creates a stand-alone copy of the current session with just the reduced data, leaving all files related to the original session unmodified.

Edit

The Edit menu allows you to launch support dialogs which control how event information is displayed.

Figure 9-8. Display Page - Edit Menu

Edit Mode

This menu choice toggles the display page mode from *edit* to *view*. While in edit mode, you can create new display page objects and change the configuration of existing ones. In view mode, you can analyze trace data.

Event Maps...

Accelerator: Ctrl+M

This menu item launches the Edit String Table dialog which allows you to change or add textual handles to event ID numbers and control which arguments are printed when event detail is shown. See Section “Edit String Table” on page 9-18 for more information.

Tags...

Accelerator: Ctrl+T

This menu choice launches the Tags dialog which lists all event tags, their time, offset and distance from the current time line, as well as any textual annotations. See Section “Tags” on page 9-14 for more information.

String Tables...

Opens the Edit String Tables dialog (see "Edit String Tables" on page 9-16) which provides a list of current string tables by name as well as the number of entries in each table. This dialog also allows you to add new string tables, and edit or remove existing string tables.

New Profile...

Accelerator: Ctrl+P

Opens the Profiles dialog allowing you to define a new profile for reference, search or summary. See Chapter 8, "Profiles" for more information.

Distinguish Processes

This menu item causes NightTrace to automatically change all process name description to include the system process (or thread) ID as part of the process name, for any process name which refers to more than one thread ID or process ID. For example, if a data set includes events from two processes name **app**, the process name description might be displayed as **app_23983** and **app_23997**.

Tags

The Tags dialog is opened by selecting the Tags... menu item (see "Tags..." on page 9-13) from the Edit menu of any display page.

The Tags dialog provides a summary of all tags in the current session. This dialog also allows for the manipulation of tags, determining time differences between two selected tags, and seeing the time difference between each tag and the *current time line* on the display page.

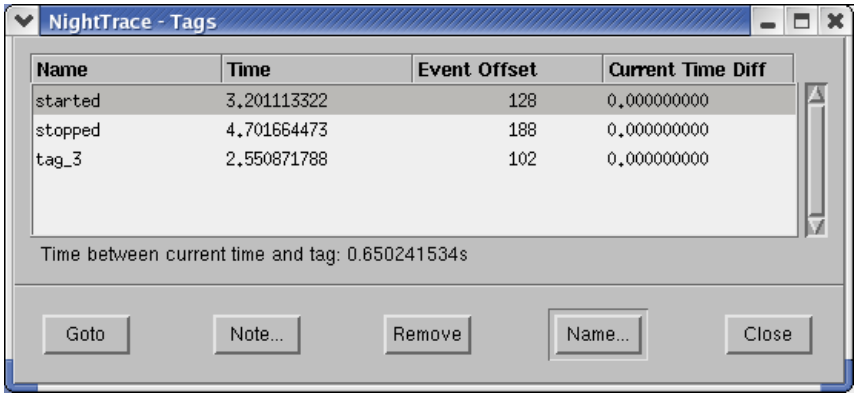


Figure 9-9. Tags dialog

To see the time difference between two tags, select the first tag of interest and then, holding the Ctrl key, select the second tag of interest. If only one tag is selected, the time difference between the selected tag and the current time is displayed.

Name

The name of the tag as entered in the Create New Tag dialog or a default name as generated by NightTrace.

Time

The time on the Ruler where the tag can be found; for tagged events, this corresponds to the event time.

Event Offset

For tagged events, this is the *offset* of the event tagged; for tagged times, this is the offset of the last event before the tag.

Current Time Diff

Shows the difference in seconds between the *current time line* on the display page and the time associated with the tag.

The following buttons appear at the bottom of the Tags dialog:

Goto

Applies to one selected tag; places the *current time line* on the selected tag on the display page; this can also be accomplished by double-clicking on the tag in the list.

Note...

Applies to one selected tag; opens the Tags Annotation dialog which allows you to add or edit notes describing the event. The notes associated with a tag are displayed in the NightTrace Main window Event Detail Area (see "Event Detail Area" on page 7-47) for the currently selected event and in a stand-alone dialog when you double click a tag marker on the "Ruler" on page 10-10 in Display Pages.

Remove

Applies to one or more selected tags; permanently removes the tag(s) from the display page.

Name

Opens the Set Tag Name dialog as shown in Figure 9-10 allowing the user to change the name of the tag selected in the Tags dialog.

Figure 9-10. Set Tag Name dialog

Close

Dismisses the Tags dialog.

Help

Displays online help for the Tags dialog.

Edit String Tables

The Edit String Tables dialog is opened by selecting the String Tables... menu item (see "String Tables..." on page 9-14) from the Edit menu of any display page. In addition, this dialog can be accessed by double-clicking the string tables icon in the NightTrace tool bar.

The Edit String Tables dialog provides a list of current string tables alphabetized by name as well as the number of entries in each table.

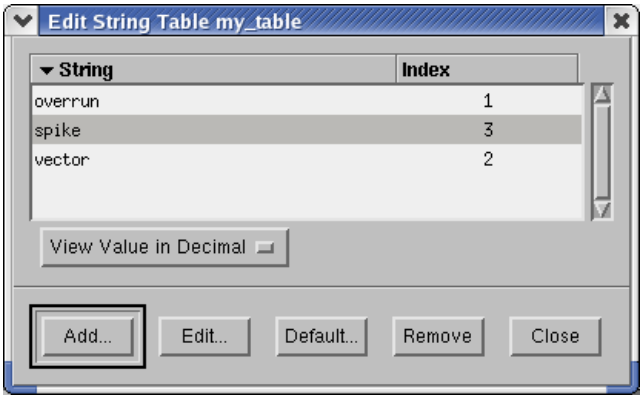


Figure 9-11. Edit String Tables dialog

Add...

Presents the Add Table dialog as shown in Figure 9-12 allowing the user to specify the name of the new string table.

Figure 9-12. Add Table dialog

After it has been added, the new string table will appear in the alphabetized list in the Edit String Tables dialog.

Edit

Opens the Edit String Table dialog (see “Edit String Table” on page 9-18) which allows you to edit the selected string table.

NOTE

You may double-click on any item in the list to edit that individual string table.

Remove

Removes the selected table(s).

NOTE

Outstanding references to removed tables remain in the session, possibly resulting in error messages in display pages, searches, and summaries.

In addition, tables populated by default by NightTrace (e.g. `event`, `boolean`) may not be removed; NightTrace silently ignores the remove request for such tables.

Save

Presents a standard file selection dialog as shown allowing the user to save the selected table(s) to an ASCII NightTrace configuration file for use in other NightTrace sessions.

NOTE

Only the selected string tables are saved.

The tables in this saved file may be imported into NightTrace by:

- specifying the file as an argument on the command line when starting NightTrace (see Chapter 6 "Invoking NightTrace")
- opening the file in a NightTrace session by selecting the Open Config File... menu choice from the NightTrace menu of the NightTrace Main window.

Close

Dismisses the Edit String Tables dialog.

Edit String Table

The Edit String Table dialog lists each string representation and its associated integer value in a particular string table and allows you to add, edit, or remove entries from that string table.

The Edit String Table dialog is opened by double-clicking the desired string table entry in the Edit String Tables dialog (see "Edit String Tables" on page 9-16) or by pressing the Edit... button while the desired string table entry is selected in the Edit String Tables dialog.

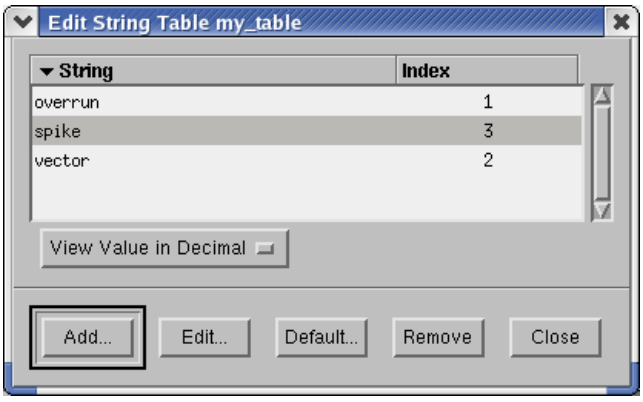


Figure 9-13. Edit String Table dialog

Two columns of are information are shown, the textual String value and the numeric Index value. Clicking the column headers change the sort key and order of the information.

View Value

Provides two options for viewing integer values in table:

View Value in Decimal

Displays each integer value in decimal representation.

View Value in Hexadecimal

Displays each integer value in hexadecimal representation.

Add...

Presents the Edit String Table Entry dialog (see “Edit String Table Entry” on page 9-20) allowing the user to add an entry to the current string table.

NOTE

When adding to the event string table, the Edit Event Map Entry dialog is presented (see “Edit Event Map Entry” on page 9-22).

Edit...

Presents the Edit String Table Entry dialog (see "Edit String Table Entry" on page 9-20) allowing the user to edit the selected entry in the current string table.

NOTE

When editing an entry in the `event` string table, the Edit Event Map Entry dialog is presented (see "Edit Event Map Entry" on page 9-22).

Default...

Presents the Edit String Table Entry dialog allowing the user to edit the default string to use when the `get_string()` function (see "get_string()" on page 11-100) is passed an integer value that is not mapped to a string in the table.

Remove

Permanently removes selected table entries.

Close

Dismisses the Edit String Table dialog.

Edit String Table Entry

The Edit String Table Entry dialog allows the user to add a new entry or edit an existing entry in a particular string table.

See "Edit String Table" on page 9-18 to add, edit, or remove other entries in a string table.

NOTE

When adding or editing an entry in the `entry` string table, NightTrace presents the Edit Event Map Entry dialog (see "Edit Event Map Entry" on page 9-22).

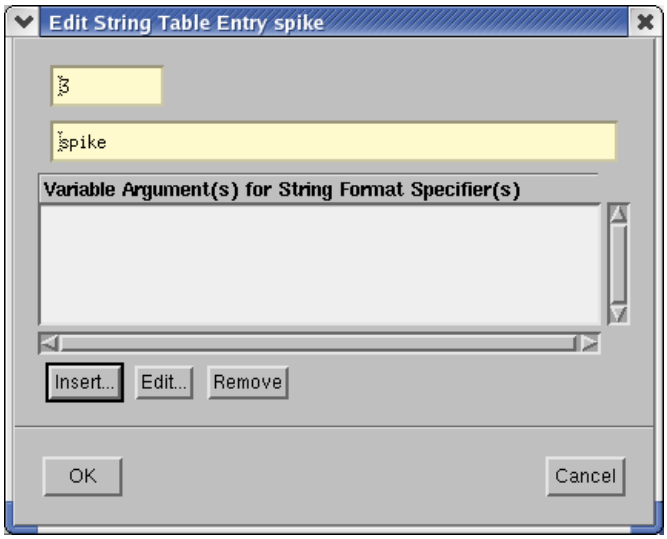


Figure 9-14. Edit String Table Entry dialog

Index

Numerical value of the string table entry.

String

String representation for the entry value.

The specified string may be a formatted string with format specifiers (see “get_format()” on page 11-104). Arguments associated with these format specifiers are added to this list using the Insert button on this dialog.

Figure 9-14 shows a String with a %d format specifer. The NightTrace expression (see “NightTrace allows you to use expressions to aid in the analysis of trace data.” on page 11-1) associated with that specifier, `offset`, can be seen in the Variable Argument(s) for String Format Specifier(s) list.

Insert

Presents the Variable Argument dialog as shown in Figure 9-15 allowing the user to associate a NightTrace expression (see “NightTrace allows you to use expressions to aid in the analysis of trace data.” on page 11-1) with a format specifier used in the String.

Figure 9-14 shows a String with a %d format specifer. The argument associated with that specifier, `offset`, can be seen in the Variable Argument(s) for String Format Specifier(s) list.

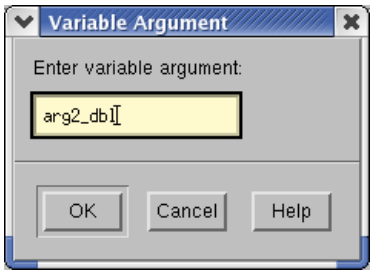


Figure 9-15. Variable Argument dialog

Edit

Presents the Variable Argument dialog as shown in Figure 9-15 allowing the user to edit the NightTrace expression (see "NightTrace allows you to use expressions to aid in the analysis of trace data." on page 11-1) selected in the Variable Argument(s) for String Format Specifier(s) list.

Remove

Removes the selected argument from the Variable Argument(s) for String Format Specifier(s) list.

Edit Event Map Entry

The Edit Event Map Entry dialog allows the user to add a new entry or edit an existing entry in the event string table.

See "Edit String Table" on page 9-18 to add, edit, or remove other entries in a string table.

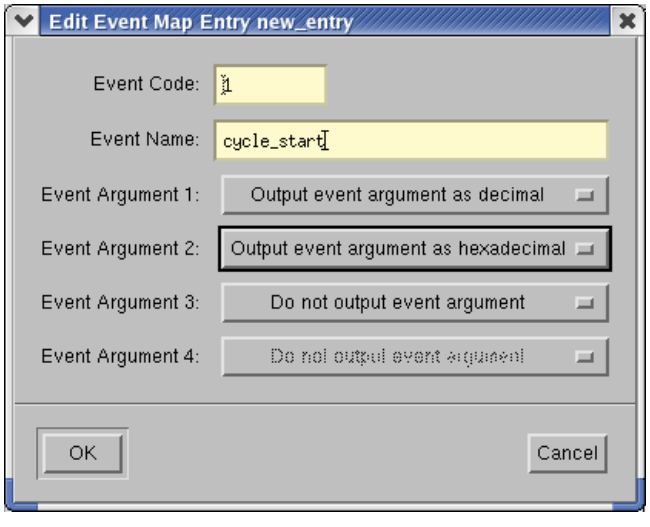


Figure 9-16. Edit Event Map Entry dialog

Event Code

A valid integer in the range reserved for user trace events (0-4095, inclusive).

Event Name

A character string to be associated with the user trace event specified in Event Code.

Trace event names must begin with a letter and consist solely of alphanumeric characters and underscores.

In addition, the user may specify up to four arguments to display for the user defined trace event. These arguments are displayed when:

- **`ntrace`** is invoked with the **`--listing`** (**`-l`**) option (see "`-l --listing`" on page 6-2)
- the user middle clicks on an event in the display page grid (shown in the Message Display Area)
- the Event Detail Area is shown on the NightTrace Main window
- when an event is found via the search mechanism (see "Search/Close" on page 8-17), the event description is given in the Message Display Area (see "Message Display Area" on page 9-29)

The following items allow the user to choose the base format in which to display each argument. The format is specified by choosing one of the following from the drop-down associated with that argument:

- Output event argument as float
- Output event argument as decimal
- Output event argument as hexadecimal
- Output event argument as float

Event Argument 1

Specifies whether the first argument is to be displayed and the base format to display that argument.

Event Argument 2

Specifies whether the second argument is to be displayed and the base format to display that argument.

Event Argument 3

Specifies whether the third argument is to be displayed and the base format to display that argument.

Event Argument 4

Specifies whether the fourth argument is to be displayed and the base format to display that argument.

Each combo box only allows for reasonable input as per the NightTrace API calls (see "Understanding NightTrace Library Calls" on page 2-4). For instance:

- if an event argument is displayed, all prior arguments must be displayed (i.e. cannot display Event Argument 2 if you are not displaying Event Argument 1)
- if Event Argument 1 is output as float, Event Argument 2 must be output as float or not output
- if Event Argument 2 is output as float, Event Argument 1 must be output as float

Zoom

The Zoom menu allows you to zoom in and out and controls the zoom factor.

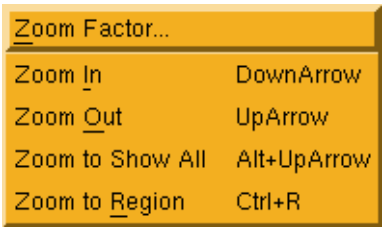


Figure 9-17. Display Page - Zoom Menu

Zoom Factor...

The menu choice allows you to set the zoom factor for subsequent Zoom In and Zoom Out operations. The factor is entered as a floating point number; the zoom factor must be a value greater than 1.0 and represents the additionl number of events to be shown in the interval on a Zoom Out operation (factored by the current number of events in the interval), and conversely on a Zoom In operation.

Zoom In

Accelerator: DownArrow

This menu choice zooms in by the current Zoom Factor.

Zoom Out

Accelerator: UpArrow

This menu choice zooms out by the current Zoom Factor.

Zoom To Show All

Accelerator: Alt+UpArrow

The menu choice zooms all the way out, displaying all events in the data set.

Zoom to Region

Accelerator: Ctrl+R

The menu choice changes the interval to include the events defined by the current *region*. The current region is defined by the events bounded by the current Mark (see “Mark” on page 9-12) and the current timeline.

View

The View menu controls which areas of the Display Page are visible and allows you to scroll the grid area up and down.

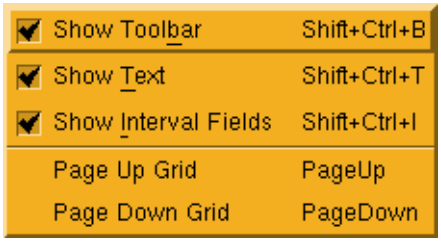


Figure 9-18. Display Page - View Menu

Show Toolbar

Acceleartor: Ctrl+Shift+B

This menu choice toggles the visibility of the Display Page .

Show Text

Acceleartor: Ctrl+Shift+T

This menu choice toggles the visibility of the Display Page Message Display Area.

Show Interval Fields

Acceleartor: Ctrl+Shift+I

This menu choice toggles the visibility of the Display Page Interval Control Area.

Page Up Grid

Acceleartor: PageUp

This menu choice scrolls the display area of the grid up.

Page Down Grid

Acceleartor: PageDown

This menu choice scrolls the display area of the grid down.

Help

The Help menu allows you to obtain help on any portion of the Display Page window or on other topics. The menu is common between the NightTrace Main window, the Profiles dialog, and Display Page windows. See "Help" on page 7-41 for more information.

Display Page Tool Bar

The Display Page tool bar provides icons for commonly used actions.

Figure 9-19. Display Page Tool Bar


Save the current NightTrace session


Search backward


Search forward


Summarize


Open profile dialog


Switch between *view* and *edit* mode


Go to event


Move timeline to previous location


Move timeline to preceeding event


Move timeline to next event


Adjust the interval to center the current timeline


Create a tag at the current timeline


Set the Mark to the current timeline

Discard events

Open Tags dialog

Open the Edit Event Map dialog

Open the Edit String Tables dialog

Zoom in

Zoom out

Zoom region

The tool bar may be hidden by clearing the Show Toolbar checkbox in the View menu of the Display Page window. Alternatively, the keyboard sequence Shift+Ctrl+B toggles the visibility of the tool bar.

Message Display Area

The Message Display Area presents various diagnostic and informational messages. Figure 9-20 shows some of these types of messages in a Message Display Area.

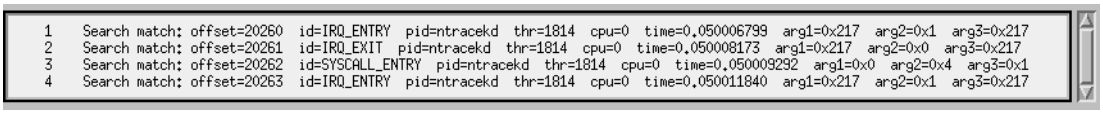


Figure 9-20. Message Display Area

The Message Display Area can include such messages as:

- error messages (e.g. from incorrect values entered in configuration dialogs)
- detailed textual information about specific events (see “Grid” on page 9-29)
- the time between the current time line and the mouse cursor (by pressing mouse button 3 at a particular point on the grid)
- the time between the mouse cursor and the mark (see “Mark” on page 9-12)
- results of search operations (see “Search/Close” on page 8-17)
- results of summary operations (see “Summarizing Statistical Information” on page 8-19)

Grid

The *grid* is a region of the display page that is filled with parallel rows and columns of dots. These dots serve as reference points for display-object alignment. You can alter the grid dimensions by changing the size of the display page. To change the display page size, resize your window by using features of your window manager.

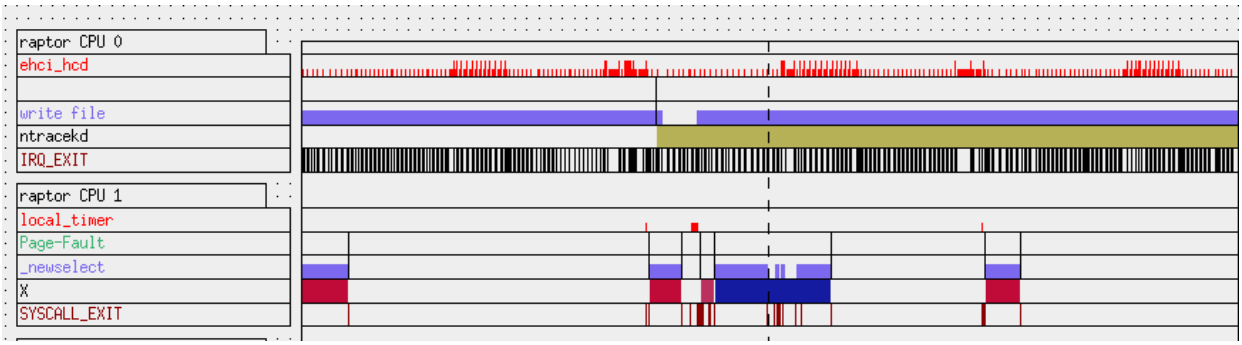


Figure 9-21. The Grid

NightTrace assigns each trace event in the trace session a unique ordinal number or *offset* beginning with ordinal number 0. These ordinal numbers appear in the interval control area and in the message display area. For more information on ordinal trace events, see "Interval Control Area" on page 9-32.

Some display objects on the grid contain vertical lines. Each vertical line in a State Graph (see "State Graph" on page 10-7) or Event Graph (see "Event Graph" on page 10-6) represents one or more user trace events, kernel trace events, or NightTrace internal trace events. If more than one event is represented by a vertical line, zooming in will provide sufficient resolution to display each trace event as a separate verticle line (see "Zoom In" on page 9-24).

If you click on a trace event with mouse button 2, NightTrace writes information about that trace event in the message display area. Each vertical line in a Data Graph (see "Data Graph" on page 10-8) represents a trace event argument. If you click on a data value with mouse button 3, NightTrace writes information about the data value in the message display area.

If your grid has a Column (see "Column" on page 10-6) and you have not already positioned your interval somewhere else, NightTrace displays in the Column the earliest 5 percent of your trace session. Usually this information is uninteresting and you want to see other parts of your trace session. The following list shows the ways you can get NightTrace to locate interesting parts of your trace session:

- Scroll through the interval using the interval scroll bar
- Zoom in or zoom out using interval push buttons
- Change the parameters defining the interval by editing its fields
- Use the Profiles dialog to search for a specific trace event or condition. (See "Profiles" on page 8-1 for more information.)

Interval Scroll Bar

Moving the slider of the interval scroll bar allows you to examine different intervals in your trace session. By moving the slider, you change the displays in display objects on the grid and in the interval control area (see “Interval Control Area” on page 9-32). Changes in the display objects are most obvious when you have a Column that contains both a State Graph and a Ruler. See Chapter 10 “Display Objects” for more information on display objects.

The interval scroll bar is horizontal and extends the entire width of the grid. The left arrowhead represents the beginning of the entire trace session, not just the part displayed on the grid or by the interval control area fields. The right arrowhead represents the end of the entire trace session.

If you have not already positioned your interval somewhere else, the movable slider of the interval scroll bar is adjacent to the scroll bar’s left arrowhead. When the slider is here, the Start Time statistic in the interval control area is 0.0000000 seconds. The length of the slider is proportionate to the amount of the trace session displayed in the interval. By default, a display page shows 5% of a trace session.

In the following interval scroll bar descriptions, the fields in the interval control area that are affected by the interval scroll bar include: Current Time, Start Time, End Time, Start Event, End Event, and Increment. For more information on these fields, see “Interval Control Area” on page 9-32.

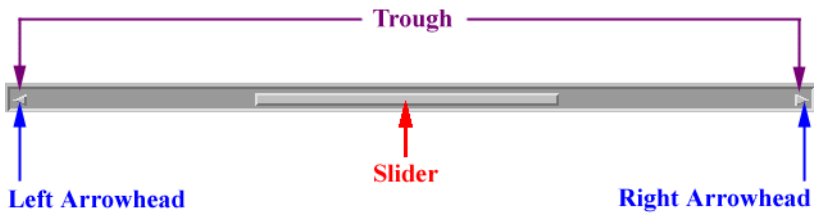


Figure 9-22. The Interval Scroll Bar

Manipulating the interval scroll bar in the following ways has the following results.

Table 9-1. Manipulating the Interval Scroll Bar

Action	Mouse Button	Location	Result
Click	Any	Left arrowhead	If the interval scroll bar slider is not already at the leftmost position: • Moves the slider to the left. • Scrolls backward Increment seconds or Increment percent of the current display interval.
Click	Any	Right arrowhead	If the interval scroll bar slider is not already at the rightmost position: • Moves the slider to the right. • Scrolls forward Increment seconds or Increment percent of the current display interval.
Click	1	Between an arrowhead and the slider	• Moves the slider to the side you clicked on. • Scrolls the current interval by twice the number of seconds in Increment or by twice the percentage in Increment.
Click or Drag	2	Between an arrowhead and the slider	• Moves the slider where you clicked and/or dragged. • Scrolls the current interval accordingly. • If your current time line was not centered, centers it.
Drag	1 or 2	Slider	(Same as preceding entry.)
Press and Hold	Any	Left or right arrowhead	Causes animated scrolling of data in the direction the arrow points

Interval Control Area

The interval control area is a region of the display page that contains nine fields of statistics. If you have not already positioned your interval somewhere else, NightTrace displays in the interval control area the earliest 5 percent of your trace session. Usually this information is uninteresting and you want to see other parts of your trace session. You can do two things with the statistics in the interval control area:

- Read the fields to obtain information about the interval
- Edit the fields to change the interval

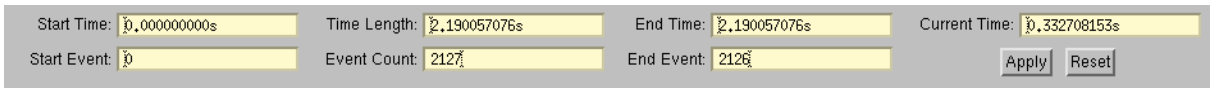


Figure 9-23. Interval Control Area

All field values in the interval control area are non-negative numbers. Some fields have default values. Time fields all display the time in seconds with the "s" suffix. A description of each field follows. In the following text, *interval* is the time from Start Time through End Time.

Start Time

The beginning time of the interval in seconds.

A valid change keeps Start Time less than the ending time in the trace session. The new interval starts at the specified time. Time Length remains unchanged, but other fields, including End Time, change appropriately.

If you set Start Time to the word `start`, NightTrace resets Start Time to the start time (0 microseconds) of the trace session.

Start Event

The ordinal number (offset), not the trace event ID, of the first trace event in this interval.

A valid change keeps Start Event less than the number of trace events logged in the trace session. The new interval starts at the specified ordinal trace event number (offset). Time Length remains unchanged, but other fields change appropriately.

If you set Start Event to the word `start`, NightTrace resets Start Event to 0 and Start Time to 0 microseconds.

Time Length

The amount of time between Start Time and End Time. Also known as the *interval*.

A valid change keeps Time Length greater than 0 and less than or equal to the last recorded time in the trace session. The new interval length is the specified length. End Time and other fields change appropriately.

If you set Time Length to the word `all` or an arbitrarily large number, NightTrace resets Time Length to the last time recorded in the trace event file(s) and changes other fields appropriately.

Event Count

The quantity of trace events present in this interval. It is the difference between End Event and Start Event plus one.

A valid change keeps Event Count less than or equal to the ordinal position (offset) of the last trace event recorded in the trace session. The new trace event count is the specified count. Fields change appropriately.

If you set Event Count to the word `all` or an arbitrarily large number, NightTrace resets Event Count to the total number of trace events in your trace event file(s) and changes other fields appropriately.

End Time

The ending time of the interval in seconds.

A valid change keeps End Time greater than the beginning time in the trace session and greater than or equal to Time Length. The new interval ends at the specified time. Time Length remains unchanged, but other fields, including Start Time, change appropriately.

If you change End Time so it is smaller than Time Length, NightTrace sets End Time to Time Length. If you set End Time to the word end or an arbitrarily large number, NightTrace resets End Time to the last time recorded in the trace event file(s) and changes other fields appropriately.

End Event

The ordinal number (offset), not the trace event ID, of the last trace event in this interval.

A valid change keeps End Event non-negative. The new interval ends at the specified ordinal trace event number (offset). Time Length remains unchanged, but other fields change appropriately.

If you set End Event to the word end, or an arbitrarily large number, NightTrace resets End Event to the total number of trace events in your trace event file(s).

Current Time

The present time within the interval in seconds.

If the new current time is inside the current interval, the current time line moves appropriately in any Columns (see "Column" on page 10-6) and the current interval remains unchanged.

If the new current time is outside the current interval, the interval shifts so the current time is centered in the interval, the current time line is centered in any Columns, and the interval length remains unchanged.

Apply

Validates any field change(s) in the interval control area (see "Interval Control Area" on page 9-32) and makes corresponding changes to other field(s), updates display objects on the grid (see "Grid" on page 9-29), and positions the current time line appropriately.

Reset

Restores changed field(s) in the interval control area (see “Interval Control Area” on page 9-32) to the value(s) they had the last time changes were applied.

10
Display Objects

A display page contains *display objects* which filter, process, and display information based upon trace event data. These display objects are created and viewed on the display page.

Display objects, which are created via the Graph menu (see "Graph" on page 9-7) on the display page, can be thought of as combination filters and formatters for the trace event data. Every time a display object is updated, it filters through the trace data. The display object accepts input in the form of a trace event record, processes and reformats the information, and displays it.

The following information is in a trace event record:

- numeric trace event ID
- global process identifier (PID)
- NightTrace thread identifier (TID)
- time
- ordinal number (offset)
- optional arguments

You can use NightTrace functions to express any of these values (see "Functions" on page 11-2).

Although trace event data contains simple events, it implicitly contains states. The concepts of trace events and states are key to understanding display objects.

trace event — Corresponds to the point in the execution of your application when a `trace_event()` call was executed. All the data logged at that time (trace event ID, arguments, etc.) is considered a trace event.

state — A state is bounded by two trace events, a *start event* and an *end event*. An instance of a state is the period of time between the start event and end event, including the start and end events themselves. Additional constraints may be specified in a state definition to further constrain the state. Instances of individual states do not nest; that is, once a state becomes active, events that might normally satisfy the conditions for the start event are ignored until the end event is encountered.

Different types of display objects display information in different ways. Depending on the type of information you want to display, you choose the display object or objects you wish to create. You can then configure those display objects to filter out unwanted data and display the information that you want.

All display objects are rectangular with user-specified dimensions and have the following properties:

- Display objects can be dynamic or static. *Dynamic* means the contents vary depending on values in the trace event file and may change depending on the *current trace event*. *Static* means the contents do not change. All display objects except Grid Labels are dynamic (see "Grid Label" on page 10-4).
- Display objects can be textual or graphical. *Textual* means the contents consist of words or numbers. *Graphical* means the contents are lines or shapes, like a bar chart.
- Display objects can be scrollable or non-scrollable. *Scrollable* means the display object acts as a movable window into the trace event file.

Types of Display Objects

The basic types of display objects are listed below and are discussed in the following sections.

- Grid Label

  Static textual display object that contains a user-specified string of text and is used to label other display objects for clarity.

  See “Grid Label” on page 10-4 for more information.

- Data Box

  Dynamic display object that displays textual or numeric information related to a trace event or state attribute associated with the current time line. The main use of a Data Box is to display data that is variable in nature and does not lend itself to graphical representation.

  See “Data Box” on page 10-5 for more information.

- Column

  Dynamic display object that does not display data itself but holds the scrollable graphical display objects: State Graphs, Event Graphs, Data Graphs, and Rulers. Its purpose is to group together related graphical display objects.

  See “Column” on page 10-6 for more information.

- Event Graph

  Dynamic, scrollable display object that graphically displays trace events as vertical lines in a Column and indicates relative chronological positions of trace events since the trace started.

  See “Event Graph” on page 10-6 for more information.

- State Graph

  Dynamic, scrollable display object that graphically displays states as bars and other trace events as vertical lines in a Column and indicates relative chronological positions of trace events and states since the trace started. This display object is usually used if you want to know when the application enters and exits a particular user-defined state.

  See “State Graph” on page 10-7 for more information.

- Data Graph

  Dynamic, scrollable display object that graphically displays numeric values as vertical lines or bars in a Column and indicates the relative chronological position of the associated trace event. The height of the line or bar is proportional to the value and is scaled according to the minimum and maximum graph values specified. This dis-

play object is commonly used to display relative values of arguments in the trace event record.

See "Data Graph" on page 10-8 for more information.

- Ruler

  Static, scrollable display object resembling a ruler that graphically displays the time. Rulers are used in a Column with State Graphs, Event Graphs, and Data Graphs to show what time a trace event occurred.

  See "Ruler" on page 10-10 for more information.

Each display page can hold multiple instances of these display objects, usually with each display object uniquely configured. All display objects on all display pages reflect the same interval and current time line; display object type, size, configuration, and position have no bearing.

Grid Label

A *Grid Label* is a rectangle that contains a string of text. This text usually is a title or description of an adjacent display object on the grid and makes the display page easier to interpret. Grid Labels can appear anywhere on the grid, but they cannot go inside a Column. You can put several Grid Labels on a grid.

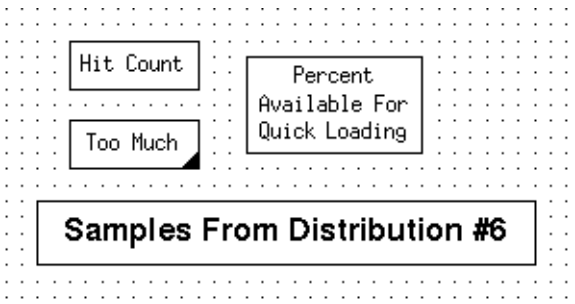


Figure 10-1. Grid Label Examples

Grid Labels are created by selecting the Grid Label menu item from the Graph menu on the display page (see "Graph" on page 9-7). See "Creating Display Objects" on page 10-12 for more information.

If the text is too long to fit into the Grid Label, the lower right corner of the box is filled in. If this occurs, you should resize the Grid Label. This is described in "Resizing Display Objects" on page 10-14. A newly created label contains the word `label`.

Grid Labels are static display objects. That is, a Grid Label does not change its appearance or contents depending on the trace event data.

In addition to specifying the text inside of the Grid Label, you also specify the color of the text (and background), the font of the text, and where in the box the text will appear (for example, top vs. bottom).

See “Grid Label” on page 10-16 for more information on configuring Grid Labels.

Data Box

A *Data Box* is a rectangle that textually displays data from the trace event file. Although the data is usually related to the last trace event received, it can also be a cumulative total or other manipulations of data in the trace event file. Data Boxes are useful when you want to display data that does not lend itself to graphical representation, as shown in Figure 10-2. This figure shows three Data Boxes: the top Data Box contains the interrupt name, the middle contains the exception name and the bottom contains the syscall name.

Figure 10-2. Data Box Examples

Data Boxes are created by selecting the Data Box menu item from the Graph menu on the display page (see “Graph” on page 9-7). See “Creating Display Objects” on page 10-12 for more information.

If a value is too large to fit into the Data Box (e.g., a long trace event name), the lower right corner of the box is filled in. If this occurs, you should resize the Data Box (see “Resizing Display Objects” on page 10-14).

By default, numeric data is displayed in decimal integer. (For information about overriding this default, see “Event Map Files” on page 6-11, “format()” on page 11-106, and “get_format()” on page 11-104.) A newly created Data Box contains a `0`.

Data Boxes can appear anywhere on the grid except within a Column. You can put several Data Boxes on a grid.

Some examples of data that you can configure a Data Box to show are:

- The name of the last trace event before the current time.
- The NightTrace thread name of the last trace event before the current time.
- A particular argument logged with the last trace event before the current time (See “arg()” on page 11-16.)
- The total amount of time the application was in a particular state before the current time (See “state_dur()” on page 11-71 and “sum()” on page 11-96.)
- The number of times a particular trace event has occurred before the current time (See “event_matches()” on page 11-35.)
- A string of characters generated by a format expression (See “format()” on page 11-106.)

See "Data Box" on page 10-18 for more information on configuring Data Boxes.

Column

A Column holds State Graphs, Event Graphs, Data Graphs and Rulers. It provides a convenient way of associating these graphical display objects. Figure 10-3 shows a Column with a Ruler added to it.

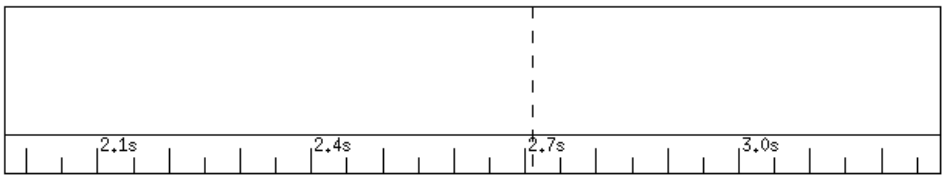


Figure 10-3. Column Example

Columns are created by selecting the Column menu item from the Graph menu on the display page (see "Graph" on page 9-7). See "Creating Display Objects" on page 10-12 for more information.

When a *Column* is first created, it is an empty rectangle that does not display data of its own.

Columns ensure that all graphical display objects within them have the same physical starting point and ending point and the same time scale. Columns are not configured, so the only variations between Columns are in their height and width.

Without a Column, you cannot put any State Graphs, Event Graphs, Data Graphs or Rulers on your grid, so you must create a Column before you can create any of these display objects.

You can place a Column anywhere on the grid. You can put more than one Column on a grid. This allows you to group related graphical objects together. All of the Columns, however, show the same interval and current time in View mode.

To hold a Ruler and any other graphical display object, Columns must be at least five grid dots high. Wider Columns are recommended because they determine the resolution to which trace events can be displayed.

Event Graph

An *Event Graph* represents trace events as a thin vertical line. Figure 10-4 shows an Event Graph with a Ruler below it.

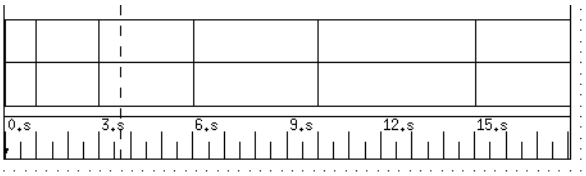


Figure 10-4. Event Graph Example

Event Graphs are created by selecting the Event Graph menu item from the Graph menu on the display page (see "Graph" on page 9-7). See "Creating Display Objects" on page 10-12 for more information.

Event Graphs must be placed in a Column (see "Column" on page 10-6).

Some examples of information that an Event Graph can be used to display are:

- The times your application starts executing a particular subroutine
- The sequence of execution of various modules in your application
- The timing of the birth and death of child processes

NOTE

In *view mode* (see "Edit Mode" on page 9-13), to find out more information about a particular trace event, position the cursor on the line and click once with mouse button 2. Information about that trace event is displayed in the message display area.

See "Event Graph" on page 10-25 for more information on configuring Event Graphs.

State Graph

A *State Graph* represents an instance of a state as a solid horizontal bar that starts when the state is active and ends when the state is inactive. A *state* is bounded by two user-specified trace events, a *start event* and an *end event*. A State Graph and a Ruler are shown in Figure 10-5.

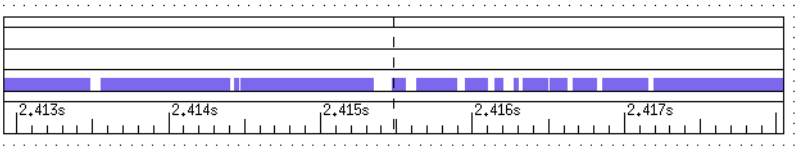


Figure 10-5. State Graph Example

State Graphs are created by selecting the State Graph menu item from the Graph menu on the display page (see "Graph" on page 9-7). See "Creating Display Objects" on page 10-12 for more information.

State Graphs must be placed in a Column (see "Column" on page 10-6).

An instance of a state is the period of time between the start event and end event, including the start and end events themselves. Instances of the same state do not nest; thus, once a state becomes active, events that might normally satisfy the conditions for the start event are ignored until the end event is encountered.

A State Graph can display trace events in a manner identical to an Event Graph. This can be useful for saving screen space or detecting when state start and state end trace events occur out of order. For example, the trace event lines can show multiple state start trace events occurring before a state end trace event.

Some examples of information that State Graphs can be used to display are:

- The times your application is executing a particular subroutine
- The differences in the execution speed of parallel threads
- The time spent in contention for resources

NOTE

In *view mode* (see "Edit Mode" on page 9-13), to find out more information about a particular trace event, position the cursor on a trace event line and click once with mouse button 2. Information about that trace event is displayed in the message display area. You can also click with mouse button 2 on the start and end of a displayed state to obtain information about the state start and state end trace events.

See "State Graph" on page 10-31 for more information on configuring State Graphs.

Data Graph

A Data Graph represents data as either vertical lines or bars of varying height. The height of the line or bar is proportional to the value specified and is scaled according to the minimum and maximum graph values allowed. This display object is usually used to display values of arguments in the trace event record.

In Figure 10-6, the same set of data is used to draw two Data Graphs which differ only by the fill style. The top Data Graph uses vertical lines of varying height to represent the data. The bottom Data Graph uses solid bars of varying height; each bar extends to the next event recognized by the Data Graph.

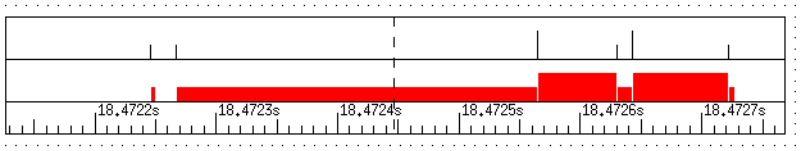


Figure 10-6. Data Graph Examples

Data Graphs are created by selecting the Data Graph menu item from the Graph menu on the display page (see “Graph” on page 9-7). See “Creating Display Objects” on page 10-12 for more information.

Data Graphs must be placed in a Column (see “Column” on page 10-6).

Some examples of ways that a Data Graph can be used are:

- track the value of an expression over time
- identify when an application variable takes on an abnormally high or low value

When choosing a size for your Data Graphs, make sure that they are high enough for you to distinguish differences in data values.

TIP

The higher you make the Data Graph, the easier it is to differentiate similar data points.

NOTE

In *view mode* (see “Edit Mode” on page 9-13), to find out about the trace event that caused the data value expression to be evaluated at a particular point, position the cursor on the line (or bar) and click once with mouse button 2. Information about the trace event is displayed in the message display area.

In *view mode*, to find out the value of a particular data item, position the cursor on the line (or bar) and click once with mouse button 3. The value of that data item is displayed in the message display area.

See “Data Graph” on page 10-37 for more information on configuring Data Graphs.

Ruler

A *Ruler* graphically displays the time interval for the current data set. Ruler display objects have major and minor hash marks to mark divisions of time since the first trace event was logged.

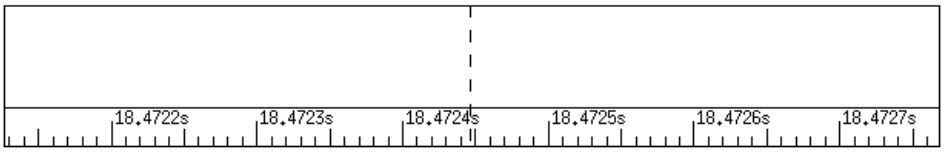


Figure 10-7. Ruler Example

Rulers are created by selecting the Ruler menu item from the Graph menu on the display page (see "Graph" on page 9-7). See "Creating Display Objects" on page 10-12 for more information.

Rulers must be placed in a Column (see "Column" on page 10-6) and should be at least three grid dots high.

In addition to hash marks and numbers, other indicators that provide useful information about the trace data being displayed are:

D	a point in time where trace event had been discarded (see "Discard Events..." on page 9-12)
L	a point in time where NightTrace lost data (see "Preventing Trace Event Loss" on page 5-1)
P	a point in time where the daemon logging trace data was paused
R	the point in time where the daemon logging trace data was resumed
?	a point in time where an erroneous timestamp was detected on a kernel trace data point
	a *mark* set by the user (see "Mark" on page 9-12)
	a *tag* set by the user (see "Tag" on page 9-12)

Figure 10-8 shows both a mark and a lost data indicator on a Ruler.

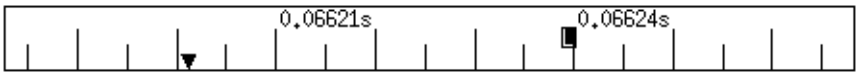


Figure 10-8. Ruler Indicators

By default, the indicators appear in reverse-video with the indicator displayed as white text over a colored background except for the mark which appears as a solid triangle. The colors of the various indicators as well as the foreground color and background color of the Ruler can be selected using the Configure Ruler dialog.

See “Ruler” on page 10-45 for more information on configuring Rulers.

Operations on Display Objects

This section describes some operations you can perform on display objects. The operations discussed are:

- Creating display objects.
- Selecting display objects.
- Moving display objects.
- Resizing display objects.
- Configuring display objects.

NOTE

The display page must be in *edit mode* in order to perform any of these operations on display objects. See "Edit Mode" on page 9-13 for more information.

Creating Display Objects

Creating display objects involves three steps: selecting the type of display object to be drawn, selecting the place on the grid where the display object will go, and selecting the size of the display object.

NOTE

The display page must be in *edit mode* in order to create display objects. See "Edit Mode" on page 9-13 for more information.

State Graphs, Event Graphs, Data Graphs and Rulers must be created inside a Column (see "Column" on page 10-6).

To create a display object and place it on the grid, do the following:

1. Select the type of display object you want to create from the Graph menu (see "Graph" on page 9-7) of the display page. (The mouse pointer changes to a crosshair).
2. Move the pointer until it is on the grid where you want to place a corner of the display object. As mentioned previously, some display objects go only inside of Columns. If the cursor is on the border of a Column or outside of one, you will not be able to draw these display objects. Note that the left and right sides of these display objects are determined by the Column, and you only have to place the pointer somewhere on the intended top or bottom edge of the display object.

3. Click and drag mouse button 1 until the display object is the size you want it to be. While you are sizing a display object, its boundaries are shown as dashed lines. Note that if you press the <Esc> key before releasing mouse button 1, the operation aborts. The display object is still loaded, as signified by the crosshair at the pointer location, so you can immediately try to recreate the display object. Also note that display objects must not overlap (except for graphical display objects, which must overlap a Column).
4. Release mouse button 1. The display object should appear on your grid with solid line boundaries, unless there was an error (e.g., you placed a Data Box on top of an existing Grid Label). Notice that the display object is also selected (corners have handles). This is in case you want to move, configure, or resize it at this time.

Selecting Display Objects

Often, you must select a display object before performing grid and edit operations. For example, before you can resize a display object you must first select the display object.

NOTE

The display page must be in *edit mode* in order to configure display objects. See "Edit Mode" on page 9-13 for more information.

To select a single display object, simply click on the display object with mouse button 1. The display object now has handles at the corners, indicating that the display object is selected.

When display objects are inside a Column, it is sometimes difficult to select the Column. To select an unselected Column, hold down the <Control> key and click mouse button 1. If you perform the same action in a selected Column, the Column is deselected.

You can select multiple display objects three different ways. The first way to select multiple display objects is as follows:

1. Position the cursor outside the display objects you want to select.
2. Click mouse button 1 and drag the mouse until the rectangle that is formed completely surrounds only the display objects you want to select. If a display object is not completely surrounded by the rectangle, it will not be selected.
3. Release mouse button 1. The display objects that were within the rectangle will now have handles at each corner.

The second way to select multiple display objects is by using the <Shift> key. Holding down the <Shift> key and clicking mouse button 1 while the cursor is in an unselected display object selects that display object without deselecting any other display objects. This allows you to select any set of display objects that you want. If you perform the same action in a display object that is already selected, the display object is deselected.

The third way to select multiple display objects is by using the Select All menu item on the Edit menu (see "Select All" on page 9-7).

Moving Display Objects

To move a display object to somewhere else on the grid, do the following:

1. Select the display object(s). Refer to "Selecting Display Objects" on page 10-13.
2. Using the mouse button 2, click anywhere on or within the selected display object(s) and drag to the desired location.
3. Release the middle button.

NOTE

The display page must be in *edit mode* in order to move display objects. See "Edit Mode" on page 9-13 for more information.

When display objects are inside a Column, it is sometimes difficult to move the Column. To move a selected Column, hold down the <Control> key and click mouse button 2.

Display objects must not overlap, although certain display objects must be placed inside a Column. If you try to move a display object on top of another display object, NightTrace displays an error message in the message display area and aborts the move.

Resizing Display Objects

To resize a display object on the grid, do the following:

1. Select the display object. See "Selecting Display Objects" on page 10-13 for more information.
2. Using mouse button 3, click on a handle and drag until the desired size is reached.
3. Release the right button.

NOTE

The display page must be in *edit mode* in order to resize display objects. See "Edit Mode" on page 9-13 for more information.

When display objects are inside a Column, it is sometimes difficult to resize the Column. To resize a selected Column, hold down the <Control> key and click mouse button 3.

Note that a Column cannot be vertically resized smaller than the minimum space required to hold all the State Graphs, Event Graphs, Data Graphs and Rulers that it contains.

Display objects must not overlap, although certain display objects must be placed inside a Column. If you try to resize a display object on top of another display object, NightTrace displays an error message in the message display area and aborts the resize.

Configuring Display Objects

Double-clicking on a particular display object will bring up the configuration dialog for that display object. In addition, you may select the Configure… menu item from the Edit menu of any display page to bring up the configuration dialog for the selected display object (see “Configure...” on page 9-8).

NOTE

The display page must be in *edit mode* in order to configure display objects. See “Edit Mode” on page 9-13 for more information.

The following sections discuss the configuration dialogs for each of the following display objects:

- Grid Label
- Data Box
- Event Graph
- State Graph
- Data Graph
- Ruler

Grid Label

The Configure Grid Label dialog is shown in Figure 10-9.

See "Grid Label" on page 10-4 for more information.

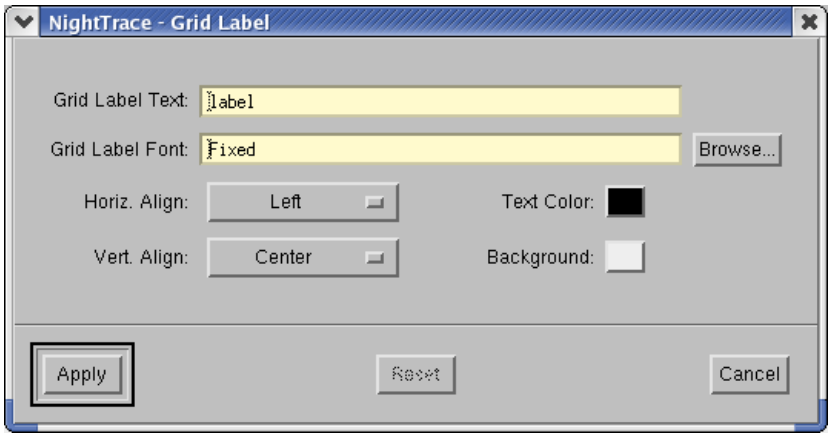


Figure 10-9. Configure Grid Label dialog

Grid Label Text

The text that is to be displayed in the Grid Label.

Grid Label Font

The font in which the Grid Label Text is to be displayed.

Browse

Presents the Choose Font dialog allowing the user to specify a font by Family, Weight, Slant, and Size.

Horizontal Alignment

Determines the justification of the text in the Grid Label.

Vertical Alignment

Determines the vertical placement of the text in the Grid Label.

Text Color

Presents the Choose Color dialog allowing the user to specify the color of the text displayed in the Grid Label. The Text Color should contrast well with the Background Color of the Grid Label.

Background Color

Presents the Choose Color dialog allowing the user to specify the color of the background of the Grid Label. The Background Color should contrast well with the Text Color.

Data Box

The Configure Data Box dialog is shown in Figure 10-10.

See "Data Box" on page 10-5 for more information.

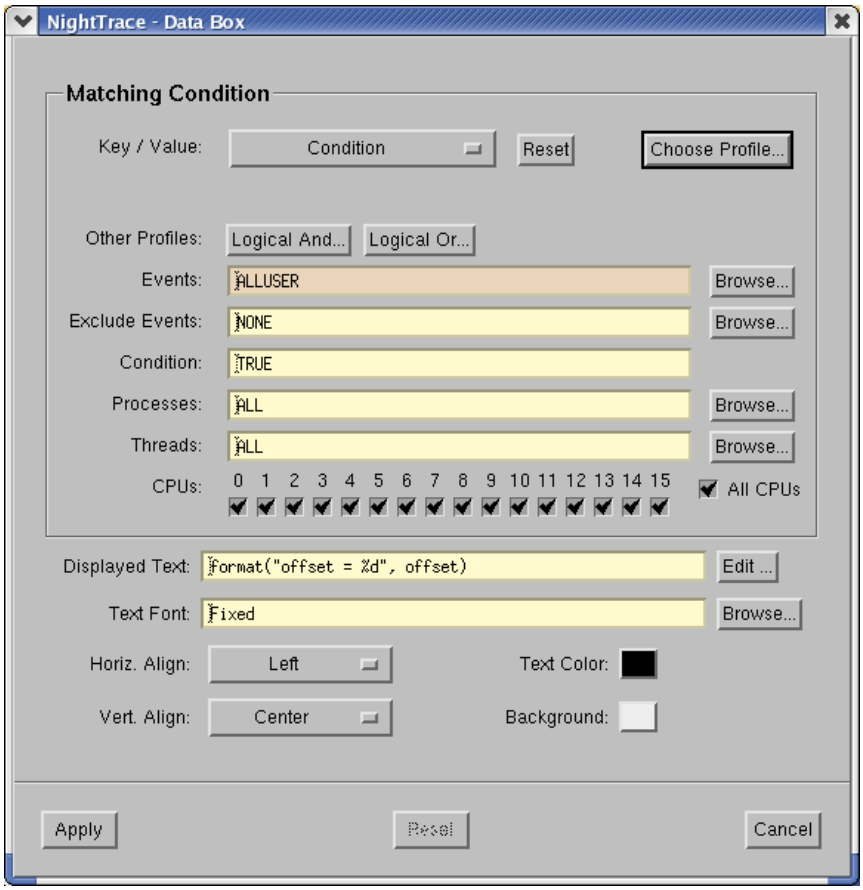


Figure 10-10. Configure Data Box dialog

Key/Value

The Key/Value option list provides a starting point for data box definition. Selecting items from the option list populates the individual criteria fields below with the values and expressions required to specify the profile you have selected.

The option list provides the following items:

Condition

This option populates the criteria fields to create a data box which will match any event, unconditionally. It is useful when you wish to manually enter criteria starting from a clean template.

System Call All Events
System Call Enter Events
System Call Exit Events

These options populate the criteria fields such that the data box 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.

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 data box can be changed by pressing the Values… button and selecting a different value from the list.

These options are desensitized if kernel trace data is not loaded.

NOTE

You can select multiple system calls from the pop-up dialog.

Exception All Events
Exception Enter Events
Exception Exit Events

These options populate the criteria fields such that the data box 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.

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 data box can be changed by pressing the Values… button and selecting a different value from the list.

These options are desensitized if kernel trace data is not loaded.

NOTE

You can select multiple exceptions from the pop-up dialog.

Interrupt All Events
Interrupt Enter Events
Interrupt Exit Events

These options populate the criteria fields such that the data box 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.

When a specific system call 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 data box can be changed by pressing the Values… button and selecting a different value from the list.

These options are desensitized if kernel trace data is not loaded.

NOTE

You can select multiple interrupts from the pop-up dialog.

Tagged Events

This option populates the criteria fields such that the data box 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 data box 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.

You can tag events with labels and annotations using the Tag icon on the tool bar, the Tags… option from the Edit dialog, as well as other actions in the NightTrace main window and in Display pages (see page 7-47 for information).

NOTE

You can select multiple tags from the pop-up dialog.

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 data box dialog with the criteria associated with that profile.

Alternatively, when checking the Import Reference to Profile checkbox in the Choose Profile… list, the dialog will be populated with an expression that references the selected profile. This technique allows you to add additional criteria within the data box while preserving the named association. Thus subsequent changes to the selected profile will be reflected in the data box.

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 criteria that were populated when you selected the profile.

Other Profiles

This area allows you to configure the data box with a condition with additional constraints associated with a previously-defined condition.

Pressing Logical And… or Logical Or… launches a list of known profiles and imports the profile you select by reference into the dialog, combining it with the current configuration via a boolean AND or OR operation, respectively.

Events

The Events criterion 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.

Exclude Events

The Exclude Events criterion 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

The Condition criterion 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 11-1 for more information on expression syntax and semantics.

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.

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 2-26 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 criterion 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"
```

CPUs

The CPUs selector area allows you to place CPU restrictions on the profile. Use the checkboxes to select the CPUs of interest.

Displayed Text

The text that is to be displayed in the Data Box when all criteria specified by this configuration dialog is met.

This is usually specified using the format() command (see "format()" on page 11-106). For example:

```
format("%s event at offset %d", get_string(event, id),offset)
```

However, you may also enter a static string by placing double quotes around the desired text.

Edit

Presents the Edit Text dialog in which to enter the output text. This dialog is useful when the desired text or format() string becomes too long to be easily edited directly in the Displayed Text field.

Text Font

The font in which the DIsplayed Text is to be displayed.

Browse

Presents the Choose Font dialog allowing the user to specify a font by Family, Weight, Slant, and Size.

Text Color

Presents the Choose Color dialog allowing the user to specify the color of the text displayed in the Data Box. The Text Color should contrast well with the Background Color of the Data Box.

Background Color

Presents the Choose Color dialog allowing the user to specify the color of the background of the Data Box. The Background Color should contrast well with the Text Color.

Horizontal Alignment

Determines the justification of the text in the Data Box.

Vertical Alignment

Determines the vertical placement of the text in the Data Box.

Event Graph

The Configure Event Graph dialog is shown in Figure 10-11.

See “Event Graph” on page 10-6 for more information.

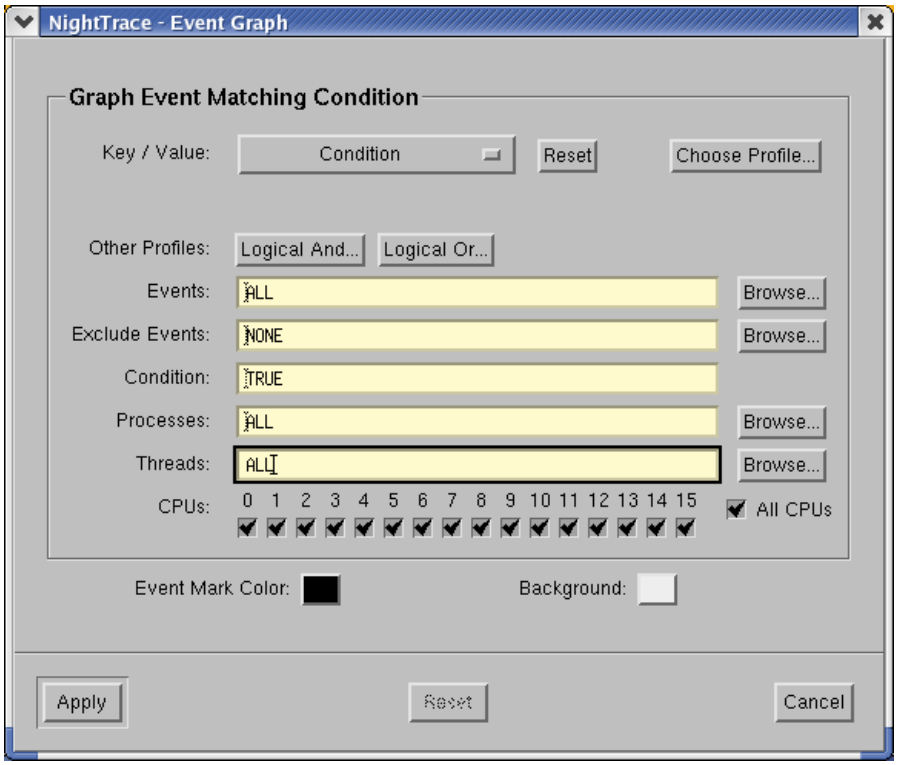


Figure 10-11. Configure Event Graph dialog

Key/Value

The Key/Value option list provides a starting point for event graph definition. Selecting items from the option list populates the individual criteria fields below with the values and expressions required to specify the profile you have selected.

The option list provides the following items:

Condition

This option populates the criteria fields to create a event graph which will match any event, unconditionally. It is useful when you wish to manually enter criteria starting from a clean template.

System Call All Events
System Call Enter Events
System Call Exit Events

These options populate the criteria fields such that the event graph 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.

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 event graph can be changed by pressing the Values… button and selecting a different value from the list.

These options are desensitized if kernel trace data is not loaded.

NOTE

You can select multiple system calls from the pop-up dialog.

Exception All Events
Exception Enter Events
Exception Exit Events

These options populate the criteria fields such that the event graph 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.

When a specific interrupt 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 event graph can be changed by pressing the Values… button and selecting a different value from the list.

These options are desensitized if kernel trace data is not loaded.

NOTE

You can select multiple exceptions from the pop-up dialog.

Interrupt All Events
Interrupt Enter Events
Interrupt Exit Events

These options populate the criteria fields such that the event graph 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.

When a specific system call 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 event graph can be changed by pressing the Values… button and selecting a different value from the list.

These options are desensitized if kernel trace data is not loaded.

NOTE

You can select multiple interrupts from the pop-up dialog.

Tagged Events

This option populates the criteria fields such that the event graph 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 event graph 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.

You can tag events with labels and annotations using the Tag icon on the tool bar, the Tags… option from the Edit dialog, as well as other actions in the NightTrace main window and in Display pages (see page 7-47 for information).

NOTE

You can select multiple tags from the pop-up dialog.

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 event graph dialog with the criteria associated with that profile.

Alternatively, when checking the Import Reference to Profile checkbox in the Choose Profile… list, the dialog will be populated with an expression that references the selected profile. This technique allows you to add additional criteria within the event graph while preserving the named association. Thus subsequent changes to the selected profile will be reflected in the event graph.

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 criteria that were populated when you selected the profile.

Other Profiles

This area allows you to configure the event graph with a condition with additional constraints associated with a previously-defined condition.

Pressing Logical And… or Logical Or… launches a list of known profiles and imports the profile you select by reference into the dialog, combining it with the current configuration via a boolean AND or OR operation, respectively.

Events

The Events criterion 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.

Exclude Events

The Exclude Events criterion 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

The Condition criterion 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 11-1 for more information on expression syntax and semantics.

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.

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 2-26 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 criterion 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"
```

CPUs

The CPUs selector area allows you to place CPU restrictions on the profile. Use the checkboxes to select the CPUs of interest.

Event Mark Color

Presents the Choose Color dialog allowing the user to specify the color of the vertical lines representing the trace events on the event graph. The Event Mark Color should contrast well with the Background Color of the Event Graph.

Background Color

Presents the Choose Color dialog allowing the user to specify the color of the background of the event graph. The Background Color should contrast well with the Event Mark Color.

State Graph

The Configure State Graph dialog is shown in Figure 10-12.

See “State Graph” on page 10-7 for more information.

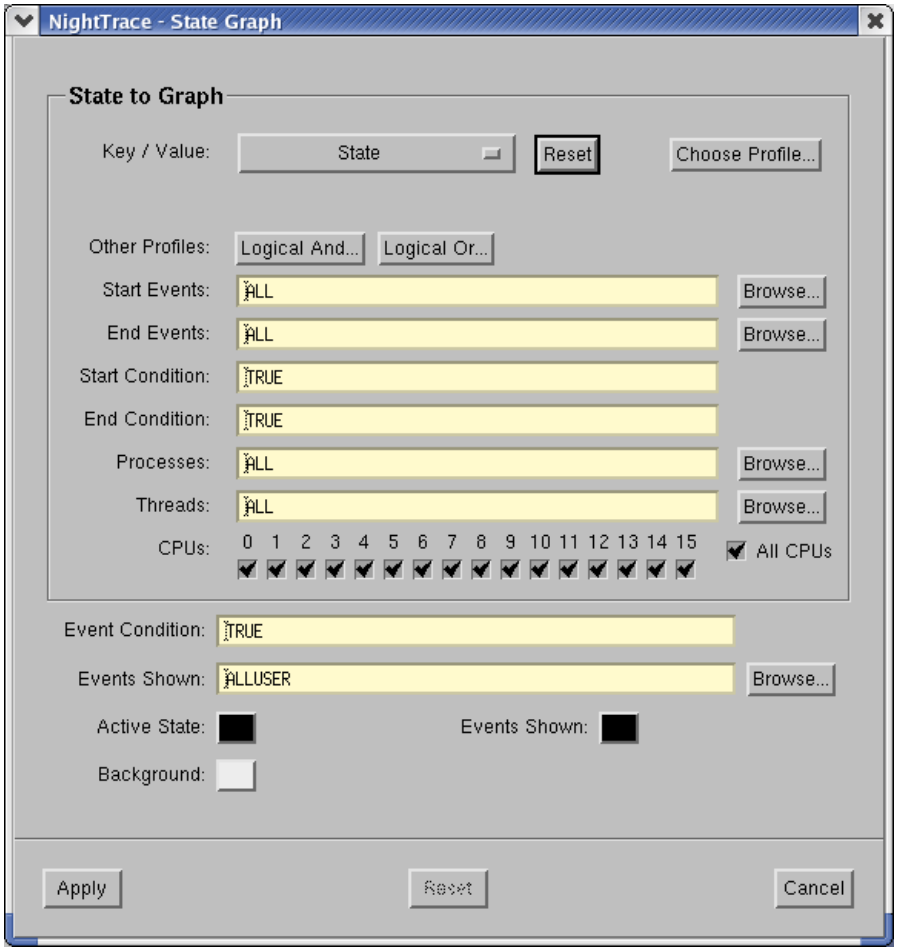


Figure 10-12. Configure State Graph dialog

A *state* is bounded by two user-specified trace events, a *start event* and an *end event*. An instance of a state is the period of time between the start event and end event, including the start and end events themselves. Instances of the same state do not nest; thus, once a state becomes active, events that might normally satisfy the conditions for the start event are ignored until the end event is encountered.

State Graphs indicate when a state is active by drawing a rectangle in the Active State Color that spans the time when the start state and end state criteria are met. In addition to drawing this state rectangle, State Graphs can behave exactly like Event Graphs by using the Events Shown. Trace event lines are superimposed on the state rectangle, which is useful for diagnosing problems where the criteria for starting the state are met multiple times before the criteria for ending the state are met.

Key/Value

The Key/Value option list provides a starting point for state graph definition. Selecting items from the option list populates the individual criteria fields below with the values and expressions required to specify the profile you have selected.

The option list provides the following items:

State

This option populates the criteria fields to create a state which starts on any event and ends on any event. It is useful when you wish to manually enter state criteria starting from a clean template.

System Call State

This option populate the criteria fields such that the state graph 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.

This option 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 state graph can be changed by pressing the Values… button and selecting a different value from the list.

This option is not present if kernel trace data is not loaded.

NOTE

You can select multiple system calls from the pop-up dialog.

Exception State

These options populate the criteria fields such that the state graph 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.

This option defines a state which begins when a 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 state graph can be changed by pressing the Values… button and selecting a different value from the list.

This option is not present if kernel trace data is not loaded.

NOTE

You can select multiple exceptions from the pop-up dialog.

Interrupt State

These options populate the criteria fields such that the state graph 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.

This option defines a state which begins when a interrupt is entered and terminates when the interrupt exits.

When a specific system call 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 state graph can be changed by pressing the Values… button and selecting a different value from the list.

This option is not present if kernel trace data is not loaded.

NOTE

You can select multiple interrupts from the pop-up dialog.

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 state graph dialog with the criteria associated with that profile.

Alternatively, when checking the Import Reference to Profile checkbox in the Choose Profile… list, the dialog will be populated with an expression that references the selected profile. This technique allows you to add additional criteria within the state graph while preserving the named association. Thus subsequent changes to the selected profile will be reflected in the state graph.

After choosing a Key/Value pair or previously defined profile using the Choose Profile… button, you can further customize the state graph 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 criteria that were populated when you selected the profile.

Other Profiles

This area allows you to configure the state graph with a condition with additional constraints associated with a previously-defined condition.

Pressing Logical And... or Logical Or... launches a list of known profiles and imports the profile you select by reference into the dialog, combining it with the current configuration via a boolean AND or OR operation, respectively.

Start Events
End Events

The 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.

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.

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 Condition
End Condition

The 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 11-1 for more information on expression syntax and semantics.

Start Condition and End Condition refers to the criteria which must be met for the beginning or end of a state, respectively.

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.

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 2-26 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 criterion 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"
```

CPUs

The CPUs selector area allows you to place CPU restrictions on the profile. Use the checkboxes to select the CPUs of interest.

Event Condition

This field specifies the condition which must evaluate to TRUE in order for an event, as defined by Events Shown, to appear. This condition does not in any way affect the Start Condition or End Condition of the state.

Events Shown

This field specifies which events will be shown in addition to the state as defined by the dialog.

Values in the text fields are required to be a comma-separated list of numeric event numbers or ranges or event names. You can use the Browse… button to the right of the text field to select from a list of known event names.

Active State Color

Presents the Choose Color dialog to allow the user to specify the color of the solid horizontal bar that represents the instance of a state in the state graph. The Active State Color should contrast well with the Background Color of the State Graph.

Events Shown Color

Presents the Choose Color dialog to allow the user to specify the color of the vertical lines that represent the events specified by Events Shown. The Events Shown Color should contrast well with the Background Color of the state graph.

Background Color

Presents the Choose Color dialog allowing the user to specify the color of the background of the state graph. The Background Color should contrast well with the Active State Color and the Events Shown Color.

Data Graph

The Configure Data Graph dialog is shown in Figure 10-13.

See “Data Graph” on page 10-8 for more information.

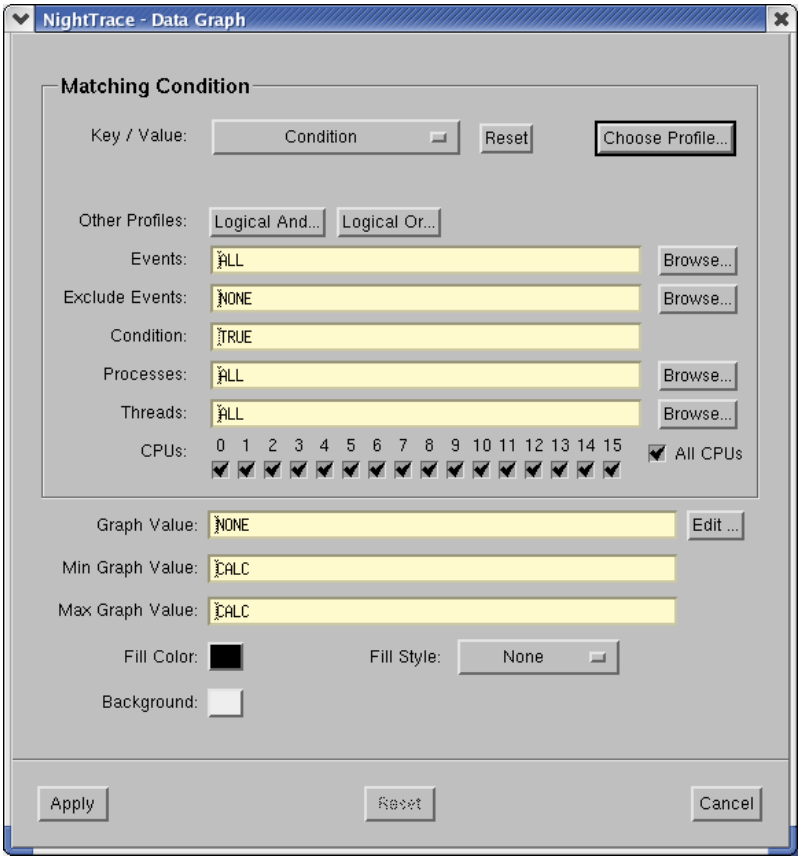


Figure 10-13. Configure Data Graph dialog

Key/Value

The Key/Value option list provides a starting point for data graph definition. Selecting items from the option list populates the individual criteria fields below with the values and expressions required to specify the profile you have selected.

The option list provides the following items:

Condition

This option populates the criteria fields to create a data graph which will match any event, unconditionally. It is useful when you wish to manually enter criteria starting from a clean template.

System Call All Events
System Call Enter Events
System Call Exit Events

These options populate the criteria fields such that the data graph 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.

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 data graph can be changed by pressing the Values… button and selecting a different value from the list.

These options are desensitized if kernel trace data is not loaded.

NOTE

You can select multiple system calls from the pop-up dialog.

Exception All Events
Exception Enter Events
Exception Exit Events

These options populate the criteria fields such that the data graph 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.

When a specific interrupt 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 data graph can be changed by pressing the Values… button and selecting a different value from the list.

These options are desensitized if kernel trace data is not loaded.

NOTE

You can select multiple exceptions from the pop-up dialog.

Interrupt All Events
Interrupt Enter Events
Interrupt Exit Events

These options populate the criteria fields such that the data graph 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.

When a specific system call 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 data graph can be changed by pressing the Values… button and selecting a different value from the list.

These options are desensitized if kernel trace data is not loaded.

NOTE

You can select multiple interrupts from the pop-up dialog.

Tagged Events

This option populates the criteria fields such that the data graph 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 data graph 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.

You can tag events with labels and annotations using the Tag icon on the tool bar, the Tags… option from the Edit dialog, as well as other actions in the NightTrace main window and in Display pages (see page 7-47 for information).

NOTE

You can select multiple tags from the pop-up dialog.

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 data graph dialog with the criteria associated with that profile.

Alternatively, when checking the Import Reference to Profile checkbox in the Choose Profile… list, the dialog will be populated with an expression that references the selected profile. This technique allows you to add additional criteria within the data graph while preserving the named association. Thus subsequent changes to the selected profile will be reflected in the data graph.

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 criteria that were populated when you selected the profile.

Other Profiles

This area allows you to configure the data graph with a condition with additional constraints associated with a previously-defined condition.

Pressing Logical And… or Logical Or… launches a list of known profiles and imports the profile you select by reference into the dialog, combining it with the current configuration via a boolean AND or OR operation, respectively.

Events

The Events criterion 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.

Exclude Events

The Exclude Events criterion 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

The Condition criterion 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 11-1 for more information on expression syntax and semantics.

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.

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 2-26 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 criterion 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"
```

CPUs

The CPUs selector area allows you to place CPU restrictions on the profile. Use the checkboxes to select the CPUs of interest.

Graph Value

The value to be graphed on the Data Graph.

The Graph Value can be any value between Min Graph Value and Max Graph Value and is usually related to the event. For instance, to graph the value of the second argument in the trace event record meeting all criteria specified by this configuration dialog, arg2 should be entered for Graph Value.

Max Graph Value

The Max Graph Value parameter determines what data value corresponds to the top of the Data Graph.

The possible values are integers or CALC. If an integer is specified as the maximum, any data that is equal to or greater than that value results in a line or bar that goes to the top of the Data Graph. If CALC is specified, the maximum value will be the greatest value found in the trace event run up to that point in time. Note that the maximum can change as time increases and new maximums are encountered.

Min Graph Value

The Min Graph Value parameter determines what data value corresponds to the bottom of the Data Graph.

The possible values are integers or CALC. If an integer is specified as the minimum, any data that is equal to or less than that value will result in no line or bar on the Data Graph. If CALC is specified, the minimum value will be the smallest value found in the trace event run up to that point in time. Note that the minimum can change as time increases and new minimums are encountered.

Fill Color

Presents the Choose Color dialog to allow the user to specify the color of the vertical line or solid horizontal bar that represents the trace event in the Data Graph when either None or Solid is selected for the Fill Style. The Fill Color should contrast well with the Background Color of the Data Graph.

Fill Style

The Fill Style parameter determines the style of Data Graph created.

The possible choices are:

None	a vertical line is drawn only at the time of a trace event
Solid	all space to the right of a trace event will be filled in the color specified by Fill Color until the next trace event is encountered
Solid by Value	all space to the right of a trace event will be filled in a color unique to the value being shown

Figure 10-14 shows the difference between Solid and None.

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

Background Color

Presents the Choose Color dialog allowing the user to specify the color of the background of the Data Graph. The Background Color should contrast well with the Fill Color.

Figure 10-15 shows the same set of data drawn in three Data Graphs, each configured differently. The data range in value from 1 to 6 and are shown at the bottom of the figure.

- The top Data Graph is configured with a minimum of 2 and a maximum of 4. Notice that several bars reach the top of the Data Graph even though they represent different data values; also note that there is no bar where data has a value less than the minimum.

- The middle Data Graph is configured with a minimum of 0 and a maximum of 10. Notice that the bars do not reach the top of the Data Graph and that the differences between values are harder to discern.
- The bottom Data Graph is configured with a minimum of 0 and a maximum set to `CALC`. Notice that the two occurrences of the maximum value of six cause bars to reach the top of the Data Graph.

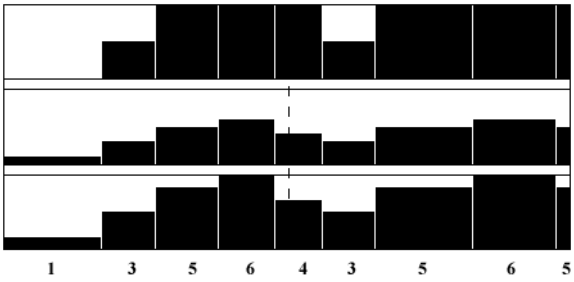


Figure 10-15. Maximum vs. Minimum Values

Ruler

The Configure Ruler dialog is shown in Figure 10-16.

See “Ruler” on page 10-10 for more information.

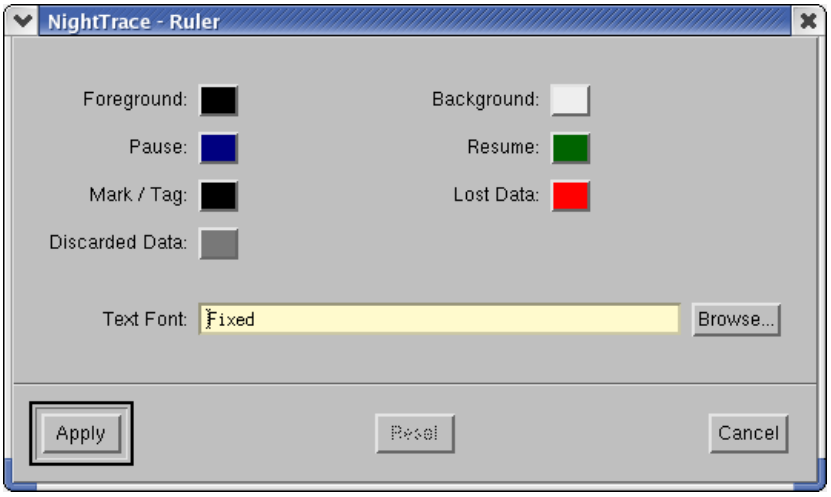


Figure 10-16. Configure Ruler dialog

Foreground Color

Presents the Choose Color dialog allowing the user to specify the color of the Ruler markings. The Foreground Color should contrast well with the Background Color.

Background Color

Presents the Choose Color dialog allowing the user to specify the color of the background of the Ruler. The Background Color should contrast well with the other items displayed on the Ruler.

Mark / Tag Color

Presents the Choose Color dialog allowing the user to specify the color of both the *mark* and *tag* indicators which appear on the Ruler. The Mark / Tag Color should contrast well with the Background Color.

Figure 10-17 shows both a mark and a tag on a Ruler

Figure 10-17. Mark and Tag Indicators

See "Mark" on page 9-12 and "Tag" on page 9-12 for more information about these indicators.

Pause Color

Presents the Choose Color dialog allowing the user to specify the color of the reverse-video "P" indicator used to show the point in time where the daemon logging trace data was paused.

Resume Color

Presents the Choose Color dialog allowing the user to specify the color of the reverse-video "R" indicator used to show the point in time where the daemon logging trace data was resumed.

Lost Data Color

The Lost Data Color specifies the color of the reverse-video "L" that is placed on a Ruler where NightTrace lost data.

See "Preventing Trace Event Loss" on page 5-1 for more information on lost data.

Discarded Data Color

Presents the Choose Color dialog allowing the user to specify the color of the reverse-video "D" indicator used to show where trace events have been discarded.

Text Font

The font in which the numeric values on the Ruler are displayed.

Browse

Presents the Choose Font dialog allowing the user to specify a font by Family, Weight, Slant, and Size.

Configuration Dialog Push Buttons

The following push buttons appear on all display object configuration dialogs.

Apply

Validate the changes you made to the configuration parameters, and apply the changes to the display object.

Reset

Discard all changes made since the last Apply.

Close

Discard any changes made since the last change was applied and close the window.

Help

Display the help topic for the display object configuration dialog.

11
Using Expressions

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 11-1 for a list of valid operators and “Operands” on page 11-1 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 casts to data types (where the parentheses are required): e.g., `(int)`

NightTrace operators follow the operator precedence rules of the C programming language.

Operands

Operands include:

- constants
- function calls
- profile references *(in functions only)*

Operand types are largely based on the C programming language and include:

- integer
- double-precision floating point
- character
- string
- boolean

Constants

Constants are one type of *operand* that may be used in NightTrace expressions.

Integer literals may be expressed using typical C language notation:

- decimal literals have no special prefix
- octal literals begin with a zero
- hexadecimal literals begin with a `0x`

Floating point literals are always considered to be double-precision floating point literals.

String literals must be enclosed within double quotes; to include a double quote in a constant string literal, precede the double quote with a backslash character. For example:

```
"possible \"meltdown\" alert"
```

The case-insensitive boolean constants `TRUE` and `FALSE` have the values `1` and `0`, respectively.

Table 11-1 shows units and suffixes for time constants.

Table 11-1. Time Units and Constant Suffixes

Time Unit	Suffix
Seconds (This is the default)	`s`
Milliseconds (10e-3 seconds)	`ms`
Microseconds (10e-6 seconds)	`us`
Nanoseconds (10e-9 seconds)	`ns`

Functions

Functions are pre-defined NightTrace entities that may be used in an *expression*. NightTrace defines five classes of functions:

- trace event functions (see 11-13)
- state functions (see 11-36)
- offset functions (see 11-74)
- summary functions (see 11-93)
- format and table functions (see 11-100)

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 11-74.

`start_` Functions with this prefix provide information about the *start event* of the *most recent instance of a state*. See “Start Functions” on page 11-36.

`end_` Functions with this prefix provide information about the *end event* of the *last completed instance of a state* See “End Functions” on page 11-53.

`state_` Functions with this prefix provide information about instances of states. See “Multi-State Functions” on page 11-70.

`event_` Functions with this prefix provide information about instances of events. See “Multi-Event Functions” on page 11-34.

Some functions can be optionally suffixed by a number, *N*, which specifies the *N*th argument logged with the trace event. *N* defaults to 1 and can have the values 1 through the maximum argument logged. For example,

`arg()` Returns the first argument

`arg1()` Returns the first argument

`arg3()` Returns the third argument

`start_id()` Returns a trace event ID

`state_gap()` Returns the time between instances of a state

Table 11-2 contains a complete list of functions.

Table 11-2. NightTrace Functions

Syntax	Return Type
`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*.
`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*.
`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.

Table 11-2. NightTrace Functions

Syntax	Return Type
`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*.
`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 11-73 for important details.
`offset` [([*PR*])] `start_offset` [([*PR*])] `end_offset` [([*PR*])]	The integer ordinal number (*offset*) of a *trace event*.

Table 11-2. NightTrace Functions

Syntax	Return Type
`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 "Profiles" on page 8-1.
offset_expr	An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.
expr	Any valid NightTrace *expression* (see "NightTrace allows you to use expressions to aid in the analysis of trace data." on page 11-1).
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 11-1 illustrates some of these concepts with a State Graph.

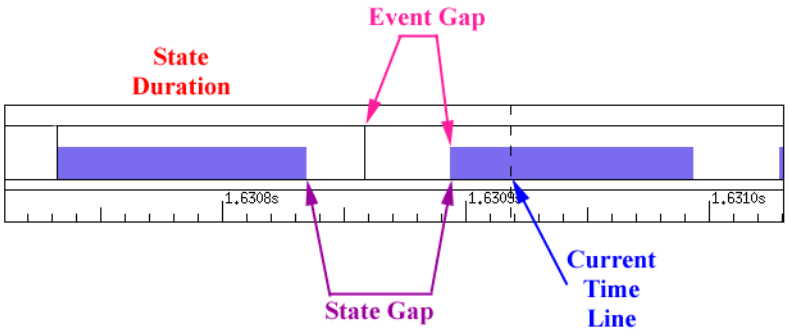


Figure 11-1. Function Terminology Illustrated

A more detailed example is illustrated in Figure 11-2.

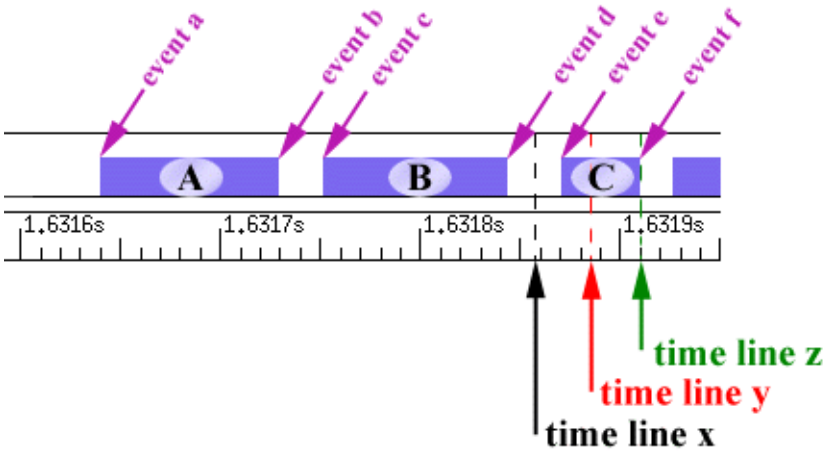


Figure 11-2. States and Events

The following discusses the terminology with respect to **time line x**, **time line y**, and **time line z**.

Assuming the current time line was positioned at **time line x** in Figure 11-2, the various "instances" would be defined as:

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 11-2.

`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 11-2, 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 11-2.

`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 11-2, 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 11-2.

`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).

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()`
- `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

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 profile. If omit-

ted, the function returns the *trace event ID* of the current trace event. For more information, see "Profile References" on page 11-107.

RETURN TYPE

integer

SEE ALSO

- "start_id()" on page 11-37
- "end_id()" on page 11-54
- "offset_id()" on page 11-75

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “start_id()” on page 11-37
- “end_id()” on page 11-54
- “offset_id()” on page 11-75

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "arg_long()" on page 11-18
- "arg_dbl()" on page 11-17
- "num_args()" on page 11-19
- "start_arg()" on page 11-38
- "end_arg()" on page 11-55
- "offset_arg()" on page 11-76

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- “arg()” on page 11-16
- “arg_long()” on page 11-18
- “num_args()” on page 11-19
- “start_arg_dbl()” on page 11-39
- “end_arg_dbl()” on page 11-56
- “offset_arg_dbl()” on page 11-77

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- "arg()" on page 11-16
- "num_args()" on page 11-19
- "start_arg_long()" on page 11-40
- "end_arg_long()" on page 11-57
- "offset_arg_long()" on page 11-78

num_args()

DESCRIPTION

The `num_args()` function returns the number of arguments logged with a *trace event*.

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “arg()” on page 11-16
- “start_num_args()” on page 11-41
- “end_num_args()” on page 11-58
- “offset_num_args()” on page 11-79

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "start_pid()" on page 11-42
- "end_pid()" on page 11-59
- "offset_pid()" on page 11-80

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “start_thread_id()” on page 11-43
- “end_thread_id()” on page 11-60
- “offset_thread_id()” on page 11-81

task_id()

DESCRIPTION

The t`ask_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 11-107.

RETURN TYPE

integer

SEE ALSO

- "start_task_id()" on page 11-44
- "end_task_id()" on page 11-61
- "offset_task_id()" on page 11-82

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “start_tid()” on page 11-45
- “end_tid()” on page 11-62
- “offset_tid()” on page 11-83

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.

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "start_cpu()" on page 11-46
- "end_cpu()" on page 11-63
- "offset_cpu()" on page 11-84

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “start_offset()” on page 11-47
- “end_offset()” on page 11-64
- “min_offset()” on page 11-97
- “max_offset()” on page 11-98

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- "event_gap()" on page 11-34
- "start_time()" on page 11-48
- "end_time()" on page 11-65
- "state_gap()" on page 11-70
- "state_dur()" on page 11-71
- "offset_time()" on page 11-85

node_id()

DESCRIPTION

The `node_id()` function returns the internally-assigned *node identifier* associated with a *trace event*.

NOTE

The `node_id()` function is of limited usefulness since the node identifier is an internally-assigned integer number assigned by NightTrace. The `node_name()` function is more useful, as it returns the name of the system from which a trace event was logged. (See “node_name()” on page 11-30 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 11-107.

RETURN TYPE

integer

SEE ALSO

- “start_node_id()” on page 11-49
- “offset_node_id()” on page 11-86
- “end_node_id()” on page 11-66

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 11-107.

RETURN TYPE

string

SEE ALSO

- "start_pid_table_name()" on page 11-50
- "offset_pid_table_name()" on page 11-87
- "end_pid_table_name()" on page 11-67

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 11-107.

RETURN TYPE

string

SEE ALSO

- “start_tid_table_name()” on page 11-51
- “offset_tid_table_name()” on page 11-88

“end_tid_table_name()” on page 11-68

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 11-107.

RETURN TYPE

string

SEE ALSO

- "start_node_name()" on page 11-52
- "offset_node_name()" on page 11-89
- "end_node_name()" on page 11-69

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 11-107.

RETURN TYPE

string

SEE ALSO

- “offset_process_name()” on page 11-90

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 11-107.

RETURN TYPE

string

SEE ALSO

- "offset_task_name()" on page 11-91

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 2-26 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 11-107.

RETURN TYPE

string

SEE ALSO

- “offset_thread_name()” on page 11-92

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- "time()" on page 11-26
- "state_gap()" on page 11-70
- "state_dur()" on page 11-71

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “summary_matches()” on page 11-99

State Functions

In its simplest form, a *state* is a region of time bounded by two *trace events*. A state definition requires the specification of two trace events, a *start event* and an *end event*, respectively. Additional conditions may be specified in a state definition to further constrain the state. The state functions include the following:

- start functions (see "Start Functions" on page 11-36)
- end functions (see "End Functions" on page 11-53)
- multi-state functions (see "Multi-State Functions" on page 11-70)

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 11-53) 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_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 11-107.

RETURN TYPE

integer

SEE ALSO

- “id()” on page 11-15
- “end_id()” on page 11-54
- “offset_id()” on page 11-75

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "arg()" on page 11-16
- "start_arg_dbl()" on page 11-39
- "start_num_args()" on page 11-41
- "end_arg()" on page 11-55
- "offset_arg()" on page 11-76

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- “arg_dbl()” on page 11-17
- “start_arg()” on page 11-38
- “start_num_args()” on page 11-41
- “end_arg_dbl()” on page 11-56
- “offset_arg_dbl()” on page 11-77

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- "arg_dbl()" on page 11-17
- "start_arg()" on page 11-38
- "start_num_args()" on page 11-41
- "end_arg_dbl()" on page 11-56
- "offset_arg_long()" on page 11-78

start_num_args()

DESCRIPTION

The `start_num_args()` function returns the number of arguments associated with the *start event* of the *most recent instance of a state*.

SYNTAX

`start_num_args` [([*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 11-107.

RETURN TYPE

integer

SEE ALSO

- “start_arg()” on page 11-38
- “num_args()” on page 11-19
- “end_num_args()” on page 11-58
- “offset_num_args()” on page 11-79

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "pid()" on page 11-20
- "end_pid()" on page 11-59
- "offset_pid()" on page 11-80

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “thread_id()” on page 11-21
- “end_thread_id()” on page 11-60
- “offset_thread_id()” on page 11-81

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "task_id()" on page 11-22
- "end_task_id()" on page 11-61
- "offset_task_id()" on page 11-82

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “tid()” on page 11-23
- “end_tid()” on page 11-62
- “offset_tid()” on page 11-83

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.

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "cpu()" on page 11-24
- "end_cpu()" on page 11-63
- "offset_cpu()" on page 11-84

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “offset()” on page 11-25
- “end_offset()” on page 11-64

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- "time()" on page 11-26
- "end_time()" on page 11-65
- "state_gap()" on page 11-70
- "state_dur()" on page 11-71
- "offset_time()" on page 11-85

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “node_id()” on page 11-27
- “offset_node_id()” on page 11-86
- “end_node_id()” on page 11-66

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 11-107.

RETURN TYPE

string

SEE ALSO

- "pid_table_name()" on page 11-28
- "offset_pid_table_name()" on page 11-87
- "end_pid_table_name()" on page 11-67

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 11-107.

RETURN TYPE

string

SEE ALSO

- “tid_table_name()” on page 11-29
- “offset_tid_table_name()” on page 11-88

“end_tid_table_name()” on page 11-68

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 11-107.

RETURN TYPE

string

SEE ALSO

- "node_name()" on page 11-30
- "offset_node_name()" on page 11-89
- "end_node_name()" on page 11-69

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 11-36) provide information about the *most recent instance of a state*.

End functions include:

- `end_id()`
- `end_arg()`
- `end_arg_dbl()`
- `end_num_args()`
- `end_pid()`
- `end_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 11-107.

RETURN TYPE

integer

SEE ALSO

- "id()" on page 11-15
- "start_id()" on page 11-37
- "offset_id()" on page 11-75

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “arg()” on page 11-16
- “start_arg()” on page 11-38
- “end_arg()” on page 11-55
- “end_num_args()” on page 11-58
- “offset_arg()” on page 11-76

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- "arg_dbl()" on page 11-17
- "start_arg_dbl()" on page 11-39
- "end_num_args()" on page 11-58
- "offset_arg_dbl()" on page 11-77

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- “arg_long()” on page 11-18
- “start_arg_long()” on page 11-40
- “end_num_args()” on page 11-58
- “offset_arg_long()” on page 11-78

end_num_args()

DESCRIPTION

The `end_num_args()` function returns the number of arguments associated with the *end event* of the *last completed instance of a state*.

SYNTAX

`end_num_args` [([*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 11-107.

RETURN TYPE

integer

SEE ALSO

- "num_args()" on page 11-19
- "start_num_args()" on page 11-41
- "end_arg()" on page 11-55
- "offset_num_args()" on page 11-79

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "pid()" on page 11-20
- "start_pid()" on page 11-42
- "offset_pid()" on page 11-80

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "thread_id()" on page 11-21
- "start_thread_id()" on page 11-43
- "offset_thread_id()" on page 11-81

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “task_id()” on page 11-22
- “start_task_id()” on page 11-44
- “offset_task_id()” on page 11-82

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "tid()" on page 11-23
- "start_tid()" on page 11-45
- "offset_tid()" on page 11-83

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.

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 11-107.

RETURN TYPE

integer

SEE ALSO

- “cpu()” on page 11-24
- “start_cpu()” on page 11-46
- “offset_cpu()” on page 11-84

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "offset()" on page 11-25
- "start_offset()" on page 11-47

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- “time()” on page 11-26
- “start_time()” on page 11-48
- “state_gap()” on page 11-70
- “state_dur()” on page 11-71
- “offset_time()” on page 11-85

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "node_id()" on page 11-27
- "start_node_id()" on page 11-49
- "offset_node_id()" on page 11-86

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 11-107.

RETURN TYPE

string

SEE ALSO

- “pid_table_name()” on page 11-28
- “start_pid_table_name()” on page 11-50
- “offset_pid_table_name()” on page 11-87

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 11-107.

RETURN TYPE

string

SEE ALSO

- "tid_table_name()" on page 11-29
- "start_tid_table_name()" on page 11-51
- "offset_tid_table_name()" on page 11-88

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 11-107.

RETURN TYPE

string

SEE ALSO

- “node_name()” on page 11-30
- “start_node_name()” on page 11-52
- “offset_node_name()” on page 11-89

Multi-State Functions

Multi-state functions return information about one or more instances of a state:

- `state_gap()`
- `state_dur()`
- `state_matches()`
- `state_status()`

For restrictions on usage, see "State Graph" on page 10-31.

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- "start_time()" on page 11-48
- "end_time()" on page 11-65
- "event_gap()" on page 11-34
- "state_dur()" on page 11-71

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 11-107.

RETURN TYPE

double-precision floating point

SEE ALSO

- “state_gap()” on page 11-70

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 11-107.

RETURN TYPE

integer

SEE ALSO

- "Start Functions" on page 11-36
- "summary_matches()" on page 11-99

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 11-107.

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_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 11-25, "start_offset()" on page 11-47, "end_offset()" on page 11-64, "min_offset()" on page 11-97, and "max_offset()" on page 11-98.

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 11-15
- “start_id()” on page 11-37
- “end_id()” on page 11-54

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 11-16
- "start_arg()" on page 11-38
- "end_arg()" on page 11-55
- "offset_arg_dbl()" on page 11-77
- "offset_num_args()" on page 11-79

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 11-17
- “start_arg_dbl()” on page 11-39
- “end_arg_dbl()” on page 11-56
- “offset_arg()” on page 11-76
- “offset_num_args()” on page 11-79

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 11-18
- "start_arg_long()" on page 11-40
- "end_arg_long()" on page 11-57
- "offset_arg()" on page 11-76
- "offset_num_args()" on page 11-79

offset_num_args()

DESCRIPTION

The `offset_num_args()` function returns the number of arguments logged with the ordinal trace event (*offset*).

SYNTAX

`offset_num_args` (*offset_expr*)

PARAMETERS

offset_expr	An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

integer

SEE ALSO

- “num_args()” on page 11-19
- “start_num_args()” on page 11-41
- “end_num_args()” on page 11-58
- “offset_arg()” on page 11-76
- “offset_arg_dbl()” on page 11-77

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 11-20
- "start_pid()" on page 11-42
- "end_pid()" on page 11-59

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 11-21
- “start_thread_id()” on page 11-43
- “end_thread_id()” on page 11-60

offset_task_id()

DESCRIPTION

The `offset_task_id()` function returns the Ada task identifier from which the ordinal trace event (*offset*) was logged.

NOTE

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

SYNTAX

`offset_task_id` (*offset_expr*)

PARAMETERS

offset_expr — An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

integer

SEE ALSO

- "task_id()" on page 11-22
- "start_task_id()" on page 11-44
- "end_task_id()" on page 11-61

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 11-23
- “start_tid()” on page 11-45
- “end_tid()” on page 11-62

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.

SYNTAX

`offset_cpu` (*offset_expr*)

PARAMETERS

offset_expr	An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

integer

SEE ALSO

- "cpu()" on page 11-24
- "start_cpu()" on page 11-46
- "end_cpu()" on page 11-63

offset_time()

DESCRIPTION

The `offset_time()` function returns the time in seconds between the beginning of the trace run and the ordinal trace event (*offset*).

SYNTAX

`offset_time` (*offset_expr*)

PARAMETERS

offset_expr	An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

double-precision floating point

SEE ALSO

- "time()" on page 11-26
- "start_time()" on page 11-48
- "end_time()" on page 11-65

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 11-27
- "start_node_id()" on page 11-49
- "end_node_id()" on page 11-66

offset_pid_table_name()

DESCRIPTION

The `offset_pid_table_name()` function returns the name of the internally-assigned NightTrace process identifier table (*PID table*) for the ordinal trace event (*offset*).

SYNTAX

`offset_pid_table_name` (*offset_expr*)

PARAMETERS

offset_expr	An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

string

SEE ALSO

- “pid_table_name()” on page 11-28
- “start_pid_table_name()” on page 11-50
- “end_pid_table_name()” on page 11-67

offset_tid_table_name()

DESCRIPTION

The `offset_tid_table_name()` function returns the name of the internally-assigned NightTrace thread identifier table (*TID table*) for the ordinal trace event (*offset*).

SYNTAX

`offset_tid_table_name` (*offset_expr*)

PARAMETERS

offset_expr	An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

string

SEE ALSO

- "tid_table_name()" on page 11-29
- "start_tid_table_name()" on page 11-51
- "end_tid_table_name()" on page 11-68

offset_node_name()

DESCRIPTION

The `offset_node_name()` function returns the name of the system from which the ordinal trace event (*offset*) was logged.

SYNTAX

`offset_node_name` (*offset_expr*)

PARAMETERS

offset_expr	An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

string

SEE ALSO

- “node_name()” on page 11-30
- “start_node_name()” on page 11-52
- “end_node_name()” on page 11-69

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 11-31

offset_task_name()

DESCRIPTION

The `offset_task_name()` function returns the name of the task from which the ordinal trace event (*offset*) was logged.

NOTE

This function is only meaningful for trace events which were logged from Ada tasking programs.

SYNTAX

`offset_task_name` (*offset_expr*)

PARAMETERS

offset_expr — An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

string

SEE ALSO

- “task_name()” on page 11-32

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 11-33

Summary Functions

You usually use summary functions on the Summarize Form. Except for summary_matches(), all of these functions take another expression as a parameter. They include the following:

- min()
- max()
- avg()
- sum()
- min_offset()
- max_offset()
- summary_matches()

min()

DESCRIPTION

The min() function returns the minimum value of all occurrences of *expr* within a time range. When used in a Summarize Form, the time range is defined by that form. When used elsewhere, the time range is defined as the region starting with the first *trace event* and ending with the *current trace event*.

SYNTAX

min (*expr*)

PARAMETERS

expr	A numeric expression.

RETURN TYPE

data type of *expr*

SEE ALSO

- "Summary Functions" on page 11-93
- "Summarizing Statistical Information" on page 8-19

max()

DESCRIPTION

The `max()` function returns the maximum value of all occurrences of *expr* within a time range. When used in a Summarize Form, the time range is defined by that form. When used elsewhere, the time range is defined as the region starting with the first *trace event* and ending with the *current trace event*.

SYNTAX

`max` (*expr*)

PARAMETERS

expr	A numeric expression.

RETURN TYPE

data type of *expr*

SEE ALSO

- "Summary Functions" on page 11-93
- "Summarizing Statistical Information" on page 8-19

avg()

DESCRIPTION

The avg() function returns the average value of all occurrences of *expr* within a time range. When used in a Summarize Form, the time range is defined by that form. When used elsewhere, the time range is defined as the region starting with the first *trace event* and ending with the *current trace event*.

SYNTAX

avg (*expr*)

PARAMETERS

expr	A numeric expression.

RETURN TYPE

data type of *expr*

SEE ALSO

- "Summary Functions" on page 11-93
- "Summarizing Statistical Information" on page 8-19

sum()

DESCRIPTION

The `sum()` function returns the sum value of all occurrences of *expr* within a time range. When used in a Summarize Form, the time range is defined by that form. When used elsewhere, the time range is defined as the region starting with the first *trace event* and ending with the *current trace event*.

SYNTAX

`sum` (*expr*)

PARAMETERS

expr	A numeric expression.

RETURN TYPE

data type of *expr*

SEE ALSO

- "Summary Functions" on page 11-93
- "Summarizing Statistical Information" on page 8-19

min_offset()

DESCRIPTION

The `min_offset()` function returns the ordinal trace event (*offset*) where the minimum value of the parameter occurred for matches in the time range. Thus, if the same minimum was seen more than once, the offset corresponds to the first one seen.

SYNTAX

`min_offset` (*expr*)

PARAMETERS

expr	A numeric expression.

RETURN TYPE

integer

NOTE

There is no function that returns the trace event ID where the minimum value of the first argument occurred for all matches in the time range. You could obtain this value by nesting the functions as follows:

```
offset_id( min_offset( arg1() ) )
```

SEE ALSO

- "Summary Functions" on page 11-93
- "Summarizing Statistical Information" on page 8-19

max_offset()

DESCRIPTION

The `max_offset()` function returns the ordinal trace event (*offset*) where the maximum value of the parameter occurred for matches in the time range. Thus, if the same maximum was seen more than once, the offset corresponds to the first one seen.

SYNTAX

`max_offset` (*expr*)

PARAMETERS

expr	A numeric expression.

RETURN TYPE

integer

NOTE

There is no function that returns the trace event ID where the maximum value of the first argument occurred for all matches in the time range. You could obtain this value by nesting the functions as follows:

```
offset_id( max_offset( arg1() ) )
```

SEE ALSO

- "Summary Functions" on page 11-93
- "Summarizing Statistical Information" on page 8-19

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 11-35
- “state_matches()” on page 11-72

Format and Table Functions

The format function allows you to display a string. The table functions allow you to extract information from user-defined and pre-defined string and format tables. These functions include the following:

- `get_string()`
- `get_item()`
- `get_format()`
- `format()`

For more information about tables, see "Tables" on page 6-14 and "Kernel String Tables" on page 12-14.

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 6-17 and "Kernel String Tables" on page 12-14.

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 6-16.

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 12-14.

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 6-16):

```
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 Box" on page 10-18):

```
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 6-16.

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 6-20.

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.

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 11-13).

A profile reference is simply the name of the profile.

Profiles are created and managed using the Profiles dialog (see “Profiles” on page 8-1for more information).

12
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 display pages or combine kernel information with user-application trace information.

NightTrace operates with the all flavors of the RedHawk kernel; standard, tracing, and debug. However, in order to use kernel tracing, you must select the tracing or debug kernel at boot time from the boot-loader menu.

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 Events tab of the Daemon Definition dialog or using **-enable** 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 11-20.

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 12-14.

`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 12-14.

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 12-14.

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 12-1:

Table 12-1. 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 12-2:

Table 12-2. 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 12-3:

Table 12-3. 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

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>
log_event (int id)
{ int fd;
  int err;
  int data[4] = {1, 2, 3, 4};
  nt_trace_custom event;
  event.id = id;
  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);
};
```

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 tab of the Daemon Definition dialog or using the **-enable** command line option to **ntracekd**, e.g:

```
ntracekd --size=20M --events=+CUSTOM data-file
```

Viewing Kernel Trace Event Files

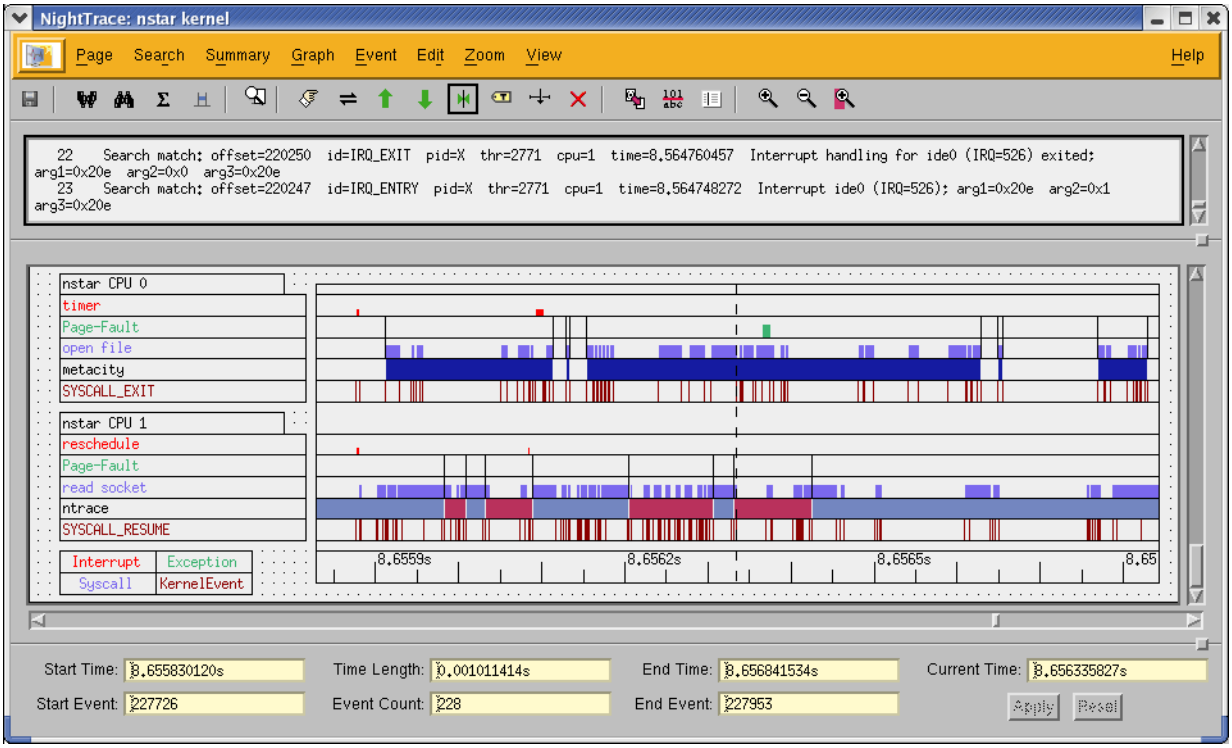
NightTrace automatically builds kernel display pages when **ntrace** is invoked with kernel data (see "Kernel Display Pages" on page 12-9). The number of CPUs is detected from the kernel trace data and controls how the page is built.

In addition, you may customize a kernel display page using the Build Custom Kernel Page dialog (see "Build Custom Kernel Page" on page 7-25) which is accessed by selecting the Custom Kernel Page… menu item from the Pages menu on the NightTrace Main Window (see "Custom Kernel Page..." on page 7-24).

Kernel Display Pages

Figure 12-1 shows a sample kernel display page for a dual CPU system.

Figure 12-1. Sample Kernel Display Page



For each CPU, several rows of information are displayed. The position of the current time line determines the values that appear on the kernel display pages. Moving the current time line within the current interval does not change the graphical displays. However, the textual displays always reflect the last values prior to or at the current time line.

The following sections discuss all of the different pieces of information in detail

Node and CPU Information

Figure 12-2 shows the Grid Label (see “Grid Label” on page 10-4) that appears on kernel display pages which displays information about the node and CPU corresponding to the trace data being displayed.

Figure 12-2. 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

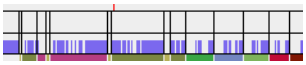


Figure 12-3. Context Switch Lines

Figure 12-3 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 12-4. Interrupt Box and Interrupt Graph

Figure 12-4 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 10-5) and the interrupt graph is a Data Graph (“Data Graph” on page 10-8). See “Configuring Display Objects” on page 10-15 for more information on configuring Data Boxes and Data Graphs.

Exception Information

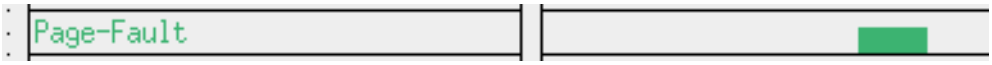


Figure 12-5. Exception Box and Exception Graph

Figure 12-5 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 12-5.

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 10-5) and the last exception graph is a State Graph (see “State Graph” on page 10-7). See “Configuring Display Objects” on page 10-15 for more information on creating and configuring Data Boxes and State Graphs.

System call Information

Figure 12-6. System Call Box and System call Graph

Figure 12-6 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 Figure 12-6.

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 10-5), and the last system call graph is a State Graph (see "State Graph" on page 10-7). See "Configuring Display Objects" on page 10-15 for more information on configuring Data Boxes and State Graphs.

Process Information

Figure 12-7. Process Information Row

Figure 12-7 shows the Process Information row which includes a process data box (see "Data Box" on page 10-5) and a process state graph (see "State Graph" on page 10-7). See "Configuring Display Objects" on page 10-15 for more information on creating and configuring Data Boxes and State Graphs.

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 12-8. Kernel Events Row

Figure 12-8 shows the Kernel Events row which includes a kernel event data box (see “Data Box” on page 10-5) and a kernel event graph (see “Event Graph” on page 10-6). See “Configuring Display Objects” on page 10-15 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 12-9. Color Key

Figure 12-9 shows the color key that is located on the bottom left of the grid on the pre-defined kernel display pages.

The text in the color key is color-coded. By default, the word “Interrupt” is red, and all display objects on the kernel display page that display information about interrupts are also red. By default, the word “Exception” is green, and all display objects that display information about exceptions are also green. By default, the word “Syscall” is blue, and all display objects that display information about 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.

The default colors of the different groups of kernel objects can be controlled with X resources. The colors are specified on a per-CPU basis. The default resources for logical CPU 0 are:

```
Ntrace*Color*GridObject*interrupt0*foreground: red
Ntrace*Color*GridObject*exception0*foreground: green
Ntrace*Color*GridObject*syscall0*foreground:   blue
Ntrace*Color*GridObject*allkernel0*foreground: darkred
```

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”)
```

`syscall_`*nodename* This string table contains the names of all possible system calls that can occur in trace event files for node name *nodename*. It is contained in the vectors file.

This table is indexed by a system call number or a system call name. Examples of using this table are:

```
get_string(syscall_systemx, 31)
get_string(syscall_systemy, arg2())
get_item(syscall_systemz, “read”)
```

`vector_`*nodename* This string table contains the interrupt and exception vector names associated with trace event files for node name *nodename*. It is contained in the vectors file.

This table is indexed by an exception/interrupt vector number or an exception/interrupt vector name. Examples of using this table are:

```
get_string(vector_machine1, arg3())
get_string(vector_machine2, 585)
get_item(vector_system3, “data access”)
```

`device_`*nodename* This string table contains the names of devices configured in the kernel for trace event files from node name *nodename*. It is contained in the vectors file.

This table is indexed by a device number or a device name. Examples of using this table are:

```
get_string(device_simulator1, arg3())
get_string(device_simulator4, 3604484)
get_item(device_controller, “rtc”)
```

The `pid` string table is also used by the kernel display pages. For more information on the `pid` string table, see “Pre-Defined Strings Tables” on page 6-17.

Part III - Programmatic Analysis

13
Using the NightTrace Analysis API

The NightTrace graphical user interface is one of the primary tools for analyzing trace data (see Chapter 7, “The NightTrace Main Window”). However, the NightTrace Analysis Application Programming Interface provides users with even further control in summarizing or monitoring trace data.

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

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

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

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

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

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

Sample programs using these data structures and functions are also provided (see NightTrace Analysis API Examples).

NightTrace Analysis Application Programming Interface

The NightTrace Analysis Application Programming Interface consists of a number of data structures (see “Data Structures” on page 13-3) and functions (see “Functions” on page 13-9).

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

`/usr/include/ntrace_analysis.h`

and the C library:

`/usr/lib/libntrace_analysis.a`

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

Data Structures

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

- `tr_cb_t` (see page 13-3)
- `tr_cond_cb_func_t` (see page 13-4)
- `tr_cond_func_t` (see page 13-4)
- `tr_cond_t` (see page 13-5)
- `tr_dir_t` (see page 13-5)
- `tr_offset_t` (see page 13-5)
- `tr_state_action_t` (see page 13-6)
- `tr_state_info_t` (see page 13-7)
- `tr_state_t` (see page 13-7)
- `tr_stream_event_t` (see page 13-8)
- `tr_string_node_t` (see page 13-8)
- `tr_t` (see page 13-8)

See “Functions” on page 13-9 for information about the functions available 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 13-3 for other data structures included in the NightTrace Analysis API.

tr_cond_cb_func_t

`tr_cond_cb_func_t` is defined as:

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

PARAMETERS

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 13-3 for other data structures included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_offset_t" on page 13-5

tr_cond_func_t

`tr_cond_func_t` is defined as:

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

See "Data Structures" on page 13-3 for other data structures included in the NightTrace Analysis API.

tr_cond_t

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

```
typedef long tr_cond_t;
```

See “Data Structures” on page 13-3 for other data structures included in the NightTrace Analysis API.

tr_dir_t

`tr_dir_t` is defined as:

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

See “Data Structures” on page 13-3 for other data structures included in the NightTrace Analysis API.

tr_offset_t

`tr_offset_t` is defined as:

```
typedef int tr_offset_t;
```

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

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

See “Data Structures” on page 13-3 for other data structures included in the NightTrace Analysis API.

tr_state_action_t

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

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

where:

`tr_state_start_action`

called for every event which starts the state

`tr_state_end_action`

called for every event which ends an active state

`tr_state_active_action`

called for every event for which the state is active

`tr_state_inactive_action`

called for every event for which the state is inactive

See "Data Structures" on page 13-3 for other data structures included in the NightTrace Analysis API.

tr_state_cb_func_t

`tr_state_cb_func_t` is defined as:

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

PARAMETERS

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 13-3 for other data structures included in the NightTrace Analysis API.

tr_state_info_t

`tr_state_info_t` is defined as:

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

where:

`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 13-3 for other data structures included in the NightTrace Analysis API.

tr_state_t

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

```
typedef long tr_state_t;
```

See “Data Structures” on page 13-3 for other data structures included in the NightTrace Analysis API.

tr_stream_event_t

`tr_stream_event_t` is defined as:

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

NOTE

The `tr_stream_overflow` event has been deprecated and no longer occurs.

See "Data Structures" on page 13-3 for other data structures included in the NightTrace Analysis API.

tr_stream_func_t

`tr_stream_func_t` is defined as:

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

See "Data Structures" on page 13-3 for other data structures included in the NightTrace Analysis API.

tr_string_node_t

`tr_string_node_t` is defined as:

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

See "Data Structures" on page 13-3 for other data structures included in the NightTrace Analysis API.

tr_t

`tr_t` is an opaque handle used to identify a particular data set. It is defined as:

```
typedef long tr_t;
```

See "Data Structures" on page 13-3 for other data structures included in the NightTrace Analysis API.

Functions

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

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

API Initialization and Destruction

The functions related to API initialization and destruction are:

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

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

SEE ALSO

- “tr_t” on page 13-8

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 13-13 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_init()" on page 13-13

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 13-15)
- `tr_error_check()` (see page 13-16)

See “Functions” on page 13-9 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 13-15 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_error_check()” on page 13-16

tr_error_check()

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

SYNTAX

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

PARAMETERS

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 13-8 for more information.)

RETURN VALUES

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

See "Error Detection, Collection, and Reporting" on page 13-15 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_string_node_t" on page 13-8
- "tr_error_clear()" on page 13-15

Input Specification and Streaming Control

The functions related to input specification and streaming control are:

- `tr_open_file()` (see page 13-17)
- `tr_open_stream()` (see page 13-18)
- `tr_close()` (see page 13-19)
- `tr_stream_notify()` (see page 13-20)
- `tr_stream_read()` (see page 13-21)
- `tr_stream_size()` (see page 13-22)
- `tr_free()` (see page 13-23)

See “Functions” on page 13-9 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 13-17 for related functions.

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

SEE ALSO

- "tr_t" on page 13-8
- "tr_close()" on page 13-19

tr_open_stream()

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

NOTE

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

SYNTAX

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

PARAMETERS

t	data set handle
fd	file descriptor providing streaming raw data
unused	this parameter is not used
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 13-17 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 13-8
- “tr_stream_size()” on page 13-22
- “tr_close()” on page 13-19

tr_close()

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

NOTE

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

SYNTAX

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

PARAMETERS

t	data set handle

See “Input Specification and Streaming Control” on page 13-17 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_open_file()” on page 13-17
- “tr_open_stream()” on page 13-18

tr_stream_notify()

`tr_stream_notify()` defines a callback which will occur when a stream event occurs as defined by `tr_stream_event_t`.

SYNTAX

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

PARAMETERS

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 13-17 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_stream_event_t" on page 13-8
- "tr_stream_func_t" on page 13-8
- "tr_stream_size()" on page 13-22
- "tr_stream_read()" on page 13-21

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 13-17 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_next_event()" on page 13-24
- "tr_iterate()" on page 13-99
- "tr_copy_input()" on page 13-94

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 13-17 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_open_stream()" on page 13-18

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 13-17 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_open_stream()” on page 13-18

Event Offset Positioning

The functions related to event offset positioning are:

- `tr_next_event()` (see page 13-24)
- `tr_next_event_()` (see page 13-25)
- `tr_prev_event()` (see page 13-25)
- `tr_prev_event_()` (see page 13-26)
- `tr_search()` (see page 13-27)
- `tr_seek()` (see page 13-28)

See "Functions" on page 13-9 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 13-24 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_offset_t" on page 13-5

tr_next_event_()

`tr_next_event_()` advances to the next consecutive trace event meeting the specified condition in the data set.

SYNTAX

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

PARAMETERS

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 13-24 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_offset_t” on page 13-5

tr_prev_event()

`tr_prev_event()` advances to the previous trace event.

SYNTAX

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

PARAMETERS

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 13-24 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_offset_t" on page 13-5

tr_prev_event_()

`tr_prev_event_()` advances to the next consecutive trace event meeting the specified condition in the data set.

SYNTAX

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

PARAMETERS

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 13-24 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_offset_t" on page 13-5

tr_search()

`tr_search()` searches for the trace event matching the specified condition in the direction specified. The current position remains unchanged.

SYNTAX

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

PARAMETERS

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 13-24 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_dir_t” on page 13-5
- “tr_cond_t” on page 13-5
- “tr_offset_t” on page 13-5

tr_seek()

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

SYNTAX

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

PARAMETERS

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 13-24 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_offset_t" on page 13-5

Basic Event Attribute Functions

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

- `tr_id()` (see page 13-30)
- `tr_id_()` (see page 13-30)
- `tr_time()` (see page 13-31)
- `tr_time_()` (see page 13-32)
- `tr_nargs()` (see page 13-32)
- `tr_nargs_()` (see page 13-33)
- `tr_arg_int()` (see page 13-34)
- `tr_arg_int_()` (see page 13-34)
- `tr_arg_dbl()` (see page 13-35)
- `tr_arg_dbl_()` (see page 13-36)
- `tr_pid()` (see page 13-36)
- `tr_pid_()` (see page 13-37)
- `tr_tid()` (see page 13-38)
- `tr_tid_()` (see page 13-38)
- `tr_thread_id()` (see page 13-39)
- `tr_thread_id_()` (see page 13-39)
- `tr_task_id()` (see page 13-40)
- `tr_task_id_()` (see page 13-41)
- `tr_cpu()` (see page 13-41)
- `tr_cpu_()` (see page 13-42)
- `tr_node()` (see page 13-43)
- `tr_node_()` (see page 13-44)
- `tr_process_name()` (see page 13-44)
- `tr_process_name_()` (see page 13-45)
- `tr_task_name()` (see page 13-46)
- `tr_task_name_()` (see page 13-46)
- `tr_thread_name()` (see page 13-47)
- `tr_thread_name_()` (see page 13-47)

See “Functions” on page 13-9 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 13-29 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8

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 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_offset_t” on page 13-5

tr_time()

`tr_time()` returns the timestamp (in seconds) of the current trace event.

NOTE

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

SYNTAX

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

PARAMETERS

t	data set handle

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8

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 13-29 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_offset_t" on page 13-5

tr_nargs()

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

SYNTAX

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

PARAMETERS

t	data set handle

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8

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.

See “Basic Event Attribute Functions” on page 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_offset_t” on page 13-5

tr_arg_int()

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

SYNTAX

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

PARAMETERS

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 13-29 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8

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 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_offset_t” on page 13-5

tr_arg_dbl()

`tr_arg_dbl()` returns the desired double argument of the current trace event.

SYNTAX

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

PARAMETERS

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 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8

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 13-29 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_offset_t" on page 13-5

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 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8

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 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_offset_t” on page 13-5

tr_tid()

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

SYNTAX

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

PARAMETERS

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 13-29 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8

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 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_offset_t” on page 13-5

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 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8

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 13-29 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_offset_t" on page 13-5

tr_task_id()

`tr_task_id()` returns the Ada task identifier associated with the current trace event.

NOTE

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

SYNTAX

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

PARAMETERS

t data set handle

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 13-29 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- “tr_t” on page 13-8

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 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_offset_t” on page 13-5

tr_cpu()

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

NOTE

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

SYNTAX

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

PARAMETERS

t	data set handle

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 13-29 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8

tr_cpu_()

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

NOTE

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

SYNTAX

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

PARAMETERS

t	data set handle
offset	offset of the trace event

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_offset_t” on page 13-5

tr_node()

`tr_node()` returns the name of the system where the current trace event was logged.

SYNTAX

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

PARAMETERS

t	data set handle

RETURN VALUES

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

See “Basic Event Attribute Functions” on page 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8

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 13-29 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_offset_t" on page 13-5

tr_process_name()

`tr_process_name()` returns the name of the process associated with the current trace event.

SYNTAX

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

PARAMETERS

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 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8

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 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_offset_t” on page 13-5

tr_task_name()

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

SYNTAX

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

PARAMETERS

t	data set handle

RETURN VALUES

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

See "Basic Event Attribute Functions" on page 13-29 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8

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 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_offset_t” on page 13-5

tr_thread_name()

`tr_thread_name()` returns the thread name associated with the current trace event.

SYNTAX

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

PARAMETERS

t	data set handle

RETURN VALUES

Returns the thread name associated with the current trace event.

See “Basic Event Attribute Functions” on page 13-29 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8

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 13-29 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_offset_t" on page 13-5

Conditions

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

- `tr_cond_create()` (see page 13-50)
- `tr_cond_reset()` (see page 13-51)
- `tr_cond_find()` (see page 13-51)
- `tr_cond_id()` (see page 13-52)
- `tr_cond_id_range()` (see page 13-53)
- `tr_cond_id_clear()` (see page 13-54)
- `tr_cond_cpu()` (see page 13-55)
- `tr_cond_cpu_clear()` (see page 13-55)
- `tr_cond_pid()` (see page 13-56)
- `tr_cond_pid_name()` (see page 13-57)
- `tr_cond_pid_clear()` (see page 13-58)
- `tr_cond_tid()` (see page 13-59)
- `tr_cond_tid_name()` (see page 13-60)
- `tr_cond_tid_clear()` (see page 13-61)
- `tr_cond_node()` (see page 13-62)
- `tr_cond_node_clear()` (see page 13-63)
- `tr_cond_func_or()` (see page 13-64)
- `tr_cond_func_and()` (see page 13-66)
- `tr_cond_func_clear()` (see page 13-68)
- `tr_cond_expr_and()` (see page 13-69)
- `tr_cond_expr_or()` (see page 13-70)
- `tr_cond_not()` (see page 13-71)
- `tr_cond_or()` (see page 13-72)
- `tr_cond_and()` (see page 13-73)
- `tr_cond_copy()` (see page 13-74)
- `tr_cond_name()` (see page 13-75)
- `tr_cond_satisfy()` (see page 13-75)
- `tr_cond_satisfy_()` (see page 13-76)
- `tr_cond_register()` (see page 13-77)

- `tr_cond_offset()` (see page 13-78)

See "Functions" on page 13-9 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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_find()" on page 13-51

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_cond_create()” on page 13-50

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_create()" on page 13-50

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_cond_create()” on page 13-50
- “tr_cond_id_range()” on page 13-53
- “tr_cond_id_clear()” on page 13-54

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_id()" on page 13-52
- "tr_cond_id_clear()" on page 13-54

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_id()" on page 13-52
- "tr_cond_id_range()" on page 13-53

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_cond_cpu_clear()” on page 13-55

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_cpu()" on page 13-55

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_cond_pid_name()” on page 13-57
- “tr_cond_pid_clear()” on page 13-58

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 13-49 for related functions.

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

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_pid()" on page 13-56
- "tr_cond_pid_clear()" on page 13-58

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_pid()" on page 13-56
- "tr_cond_pid_name()" on page 13-57

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_cond_tid_name()” on page 13-60
- “tr_cond_tid_clear()” on page 13-61

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_tid()" on page 13-59
- "tr_cond_tid_clear()" on page 13-61

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5

tr_cond_node()

`tr_cond_node()` appends the specified system node name to the list of required node names that must be matched for the given condition to evaluate to TRUE.

NOTE

Before the first `tr_cond_node()` call or after calling `tr_cond_node_clear()`, the node requirement is empty which matches any node.

SYNTAX

```
extern int tr_cond_node (tr_t t,
                         tr_cond_t cond,
                         char * node);
```

PARAMETERS

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_node_clear()" on page 13-63

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_cond_node()” on page 13-62

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	

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_cond_func_t” on page 13-4
- “tr_cond_or()” on page 13-72
- “tr_cond_and()” on page 13-73
- “tr_cond_func_and()” on page 13-66
- “tr_cond_func_clear()” on page 13-68

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	

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_cond_func_t” on page 13-4
- “tr_cond_or()” on page 13-72
- “tr_cond_and()” on page 13-73
- “tr_cond_func_or()” on page 13-64
- “tr_cond_func_and()” on page 13-66

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_func_or()" on page 13-64
- "tr_cond_func_clear()" on page 13-68

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_cond_expr_or()” on page 13-70

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_expr_and()" on page 13-69

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_cond_or()” on page 13-72
- “tr_cond_and()” on page 13-73

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_not()" on page 13-71

- "tr_cond_and()" on page 13-73

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5

- "tr_cond_not()" on page 13-71
- "tr_cond_or()" on page 13-72

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_cond_or()" on page 13-72
- "tr_cond_and()" on page 13-73

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5

tr_cond_satisfy()

`tr_cond_satisfy()` is used to determine if the current event satisfies the specified condition.

SYNTAX

```
extern int tr_cond_satisfy (tr_t t,
                            tr_cond_t cond);
```

PARAMETERS

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5

tr_cond_satisfy_()

`tr_cond_satisfy_()` is used to determine if the trace event at the specified offset satisfies the specified condition.

SYNTAX

```
extern int tr_cond_satisfy_ (tr_t t,
                             tr_cond_t cond,
                             tr_offset_t offset);
```

PARAMETERS

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5

- “tr_offset_t” on page 13-5

tr_cond_register()

`tr_cond_register()` registers the specified condition so that it is evaluated for every event.

NOTE

Registration of conditions increases processing time.

SYNTAX

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

PARAMETERS

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 13-49 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_activate()” on page 13-89
- “Profile References” on page 11-107

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 13-49 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5
- "tr_offset_t" on page 13-5

State-oriented Interfaces

The functions that deal with the creation, configuration, and activation of states are:

- `tr_state_create()` (see page 13-80)
- `tr_state_find()` (see page 13-81)
- `tr_state_name()` (see page 13-81)
- `tr_state_start_id()` (see page 13-82)
- `tr_state_start_id_range()` (see page 13-83)
- `tr_state_start_id_clear()` (see page 13-84)
- `tr_state_end_id()` (see page 13-84)
- `tr_state_end_id_range()` (see page 13-85)
- `tr_state_end_id_clear()` (see page 13-86)
- `tr_state_start_cond()` (see page 13-86)
- `tr_state_start_cond_clear()` (see page 13-87)
- `tr_state_end_cond()` (see page 13-88)
- `tr_state_end_cond_clear()` (see page 13-88)
- `tr_activate()` (see page 13-89)
- `tr_state_info()` (see page 13-90)
- `tr_state_info_()` (see page 13-91)
- `tr_state_active()` (see page 13-92)
- `tr_state_active_()` (see page 13-93)

See “Functions” on page 13-9 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 13-79 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8

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 13-79 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_state_t” on page 13-7
- “tr_state_create()” on page 13-80

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 13-79 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_state_t" on page 13-7

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 13-79 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8

- “tr_state_t” on page 13-7

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 13-79 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_state_t” on page 13-7

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 13-79 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_state_t" on page 13-7

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 13-79 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_state_t” on page 13-7

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 13-79 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8

- "tr_state_t" on page 13-7

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 13-79 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_state_t" on page 13-7

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 13-79 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_state_t” on page 13-7
- “tr_cond_t” on page 13-5

tr_state_start_cond_clear()

`tr_state_start_cond_clear()` clears any conditions associated with start of a particular state.

SYNTAX

```
extern void tr_state_start_cond_clear (tr_t t,
                                       tr_state_t state);
```

PARAMETERS

t	data set handle
state	handle of the state

See “State-oriented Interfaces” on page 13-79 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_state_t” on page 13-7

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 13-79 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_state_t" on page 13-7
- "tr_cond_t" on page 13-5

tr_state_end_cond_clear()

`tr_state_end_cond_clear()` clears any conditions associated with end of a particular state.

SYNTAX

```
extern void tr_state_end_cond_clear (tr_t t,
                                     tr_state_t state);
```

PARAMETERS

t	data set handle
state	handle of the state

See "State-oriented Interfaces" on page 13-79 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_state_t” on page 13-7

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 13-79 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_state_active()” on page 13-92

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 13-7).

If the state has never been active, `start_offset` and `end_offset` are set to `TR_EOF` and `gap` and `duration` are set to zero.

See "State-oriented Interfaces" on page 13-79 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_state_t" on page 13-7
- "tr_state_info_t" on page 13-7

tr_state_info_()

`tr_state_info_()` returns a structure containing the current values associated with the given state at the specified offset.

NOTE

Calling `tr_state_info_()` is an expensive operation if the specified offset is not the current position.

SYNTAX

```
extern void tr_state_info_ (tr_t t,
                            tr_state_t state,
                            tr_state_info_t * info,
                            tr_offset_t offset);
```

PARAMETERS

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 13-7).

If the state has never been active, `start_offset` and `end_offset` are set to `TR_EOF` and `gap` and `duration` are set to zero.

See “State-oriented Interfaces” on page 13-79 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_state_t” on page 13-7
- “tr_state_info_t” on page 13-7
- “tr_offset_t” on page 13-5

tr_state_active()

`tr_state_active()` is used to determine if the specified state is active at the current offset.

SYNTAX

```
extern int tr_state_active (tr_t t,
                            tr_state_t state);
```

PARAMETERS

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 13-79 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_state_t" on page 13-7

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 13-79 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_state_t” on page 13-7
- “tr_offset_t” on page 13-5

Output Function

The function dealing with the output of trace data is:

- `tr_copy_input()` (see page 13-94)

See "Functions" on page 13-9 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 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cond_t" on page 13-5

String Table Functions

The following functions are provided to create, manage, and search NightTrace string tables:

- `tr_get_string()` (see page 13-95)
- `tr_get_item()` (see page 13-96)
- `tr_create_table()` (see page 13-97)
- `tr_append_table()` (see page 13-98)

See “Functions” on page 13-9 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 13-95 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “String Tables” on page 6-16

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 13-95 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "String Tables" on page 6-16

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 13-95 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_string_node_t” on page 13-8
- “String Tables” on page 6-16

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 13-95 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "String Tables" on page 6-16

Callback Interfaces

The following functions deal with the callback capabilities of the NightTrace Analysis Application Programming Interface:

- `tr_iterate()` (see page 13-99)
- `tr_halt()` (see page 13-100)
- `tr_cancel_cb()` (see page 13-100)
- `tr_cond_cb()` (see page 13-101)
- `tr_state_cb()` (see page 13-102)

See “Functions” on page 13-9 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 13-99 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_cb()” on page 13-101
- “tr_state_cb()” on page 13-102
- “tr_open_stream()” on page 13-18

tr_halt()

`tr_halt()` halts the iteration process, causing `tr_iterate()` to return.

SYNTAX

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

PARAMETERS

t	data set handle

See "Callback Interfaces" on page 13-99 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_iterate()" on page 13-99

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 13-99 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_cb_t" on page 13-3

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 13-99 for related functions.

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

SEE ALSO

- “tr_t” on page 13-8
- “tr_cond_t” on page 13-5
- “tr_cond_cb_func_t” on page 13-4
- “tr_cb_t” on page 13-3

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 13-6)
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 13-99 for related functions.

See "Functions" on page 13-9 for a complete list of functions included in the NightTrace Analysis API.

SEE ALSO

- "tr_t" on page 13-8
- "tr_state_t" on page 13-7
- "tr_state_action_t" on page 13-6
- "tr_state_cb_func_t" on page 13-6
- "tr_cb_t" on page 13-3

Part IV - Reference

Part IV Reference

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.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 full 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.

- Hot operations

  Users of NightView gain the ability to read and write to a particular process without having to stop it. Thus, all eventpoints can be applied and modified during application program execution without stopping the process. User variables also can be read and modified without stopping the process.

- Signal handling

  Allows NightView to pass signals directly to a particular process, avoiding context switching.

NOTE

NightView may not operate at all on older versions of Red Hat without the RedHawk kernel.

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 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 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.

Advantages for NightSim

The following advantage is afforded NightSim when Concurrent's RedHawk kernel is running:

- Scheduling target

  Allows NightSim to schedule processes on the system via Concurrent's Frequency-Based Scheduler.

C
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:


State Summary Results
```

```
====================

Number of states found:       100

Maximum state duration:       0.000027722 at offset: 79
Minimum state duration:       0.000012817 at offset: 5
Average state duration:       0.000014569
Total of state durations:     0.001456897

Number of state gaps found:   100

Maximum state gap:            0.000000430 at offset: 3
Minimum state gap:            0.000000303 at offset: 13
Average state gap:            0.000000306
Total of state gaps:          0.000030604
```

Multi-Threaded C++ Example

This example uses demonstrates using NightTrace event logging from multiple threads.

```
#include <stdlib.h>
#include <ntrace.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_register_thread();
   trace_open_thread ("romeo");
   while (!done) {
      trace_event_arg (Start, i);
      result = work(i++);
      trace_event_arg(End, result);
   }
}

void *
thread_b (void * ptr)
{
   int i=9999999;
   int result;
   trace_register_thread();
   trace_open_thread ("juliet");
   while (!done) {
      trace_event_arg (Start, i);
      result = work(i--);
      trace_event_arg(End, result);
   }
}

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 registers itself with the NightTrace API via a `trace_register_thread`() call, and then identifies itself with a unique name via the `trace_open_thread`() call.

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 2-26 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
```

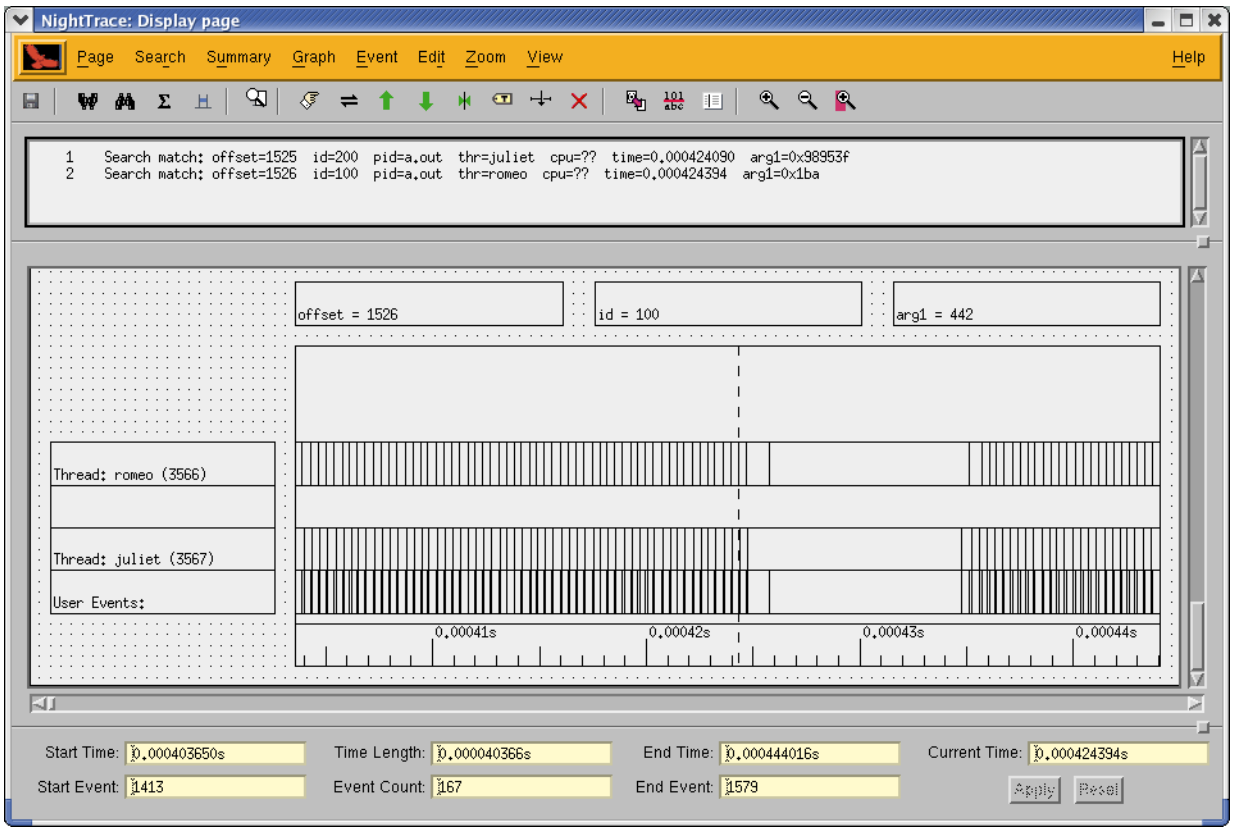


Figure 1-1. Automatically Generated Data Display Page

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_start("data")
      void = trace_open_thread("fmain")

      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=fmain        time=0.000000000s arg1=0x1
1: cpu=?? 1  pid=a.out     thr=fmain        time=0.000002481s arg1=0x2
2: cpu=?? 1  pid=a.out     thr=fmain        time=0.000003103s arg1=0x3
3: cpu=?? 1  pid=a.out     thr=fmain        time=0.000003536s arg1=0x4
4: cpu=?? 1  pid=a.out     thr=fmain        time=0.000003976s arg1=0x5
5: cpu=?? 1  pid=a.out     thr=fmain        time=0.000004386s arg1=0x6
6: cpu=?? 1  pid=a.out     thr=fmain        time=0.000004882s arg1=0x7
7: cpu=?? 1  pid=a.out     thr=fmain        time=0.000005302s arg1=0x8
8: cpu=?? 1  pid=a.out     thr=fmain        time=0.000005820s arg1=0x9
9: cpu=?? 1  pid=a.out     thr=fmain        time=0.000006294s arg1=0xa
...
```

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>

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.

D
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 13-1).

NOTE

The source files for these programs are installed in **`/usr/lib/NightTrace/examples`**.

- **`list`** (see “list” on page D-2)

  This program simply lists each NightTrace event using a simple main loop to position to the next event.

- **`search`** (see “search” on page D-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 D-7)

  This program illustrates how to monitor a certain condition in real-time and then act upon it accordingly.

- **`ptime`** (see “ptime” on page D-10)

  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 D-13)

  This program contains a collection of code segments which might be useful for reference.

- **`detect`** (see “detect” on page D-24)

  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 D-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 11-1) enclosed by double quotes.

The **`search`** program builds a *condition* object and assigns the specified expression to that condition. It then registers a callback to the `print` function for every event that satisfies the *condition*. It then invokes the `iterate` function to process the entire *trace_data_file.*

To call the **`search`** program with a *trace_data_file* named **`my_trace_data`** and the *NightTrace_Expression*:

`num_args>1 && arg2==0`

you would issue the following command:

`./search my_trace_data "num_args>1 && arg2==0"`

See "NightTrace Analysis API Examples" on page D-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*.

For simplicity, this program only lists the time at which the context switch occurred and the process being switched in.

This program may be invoked with the following command:

```
ntracekd --stream /tmp/handle | ./watchdog 1
```

or it can be launched from the NightTrace GUI as part of a streaming kernel daemon definition (via the setting of the Stream checkbox on the General page of the Daemon Definition dialog (see “Stream” on page 7-64)).

See “NightTrace Analysis API Examples” on page D-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 D-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 D-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 (via the setting of the Stream checkbox on the General page of the Daemon Definition dialog (see "Stream" on page 7-64)).

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 D-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);
}
```

E
Answers to Common Questions

Q: What can I do if trace events are not logging at all?

A: Verify that the trace event file name on the `trace_begin()` call matches the one on the user daemon invocation. Furthermore, check that the file exists and that you have permission to read and write it. Check the return codes from the API calls. Additionally, be sure your thread name, if specified to `trace_open_thread()` contains no embedded spaces or punctuation, including periods. See "trace_begin" on page 2-6 and "trace_open_thread" on page 2-11 for more information.

Q: When should I log a different trace event ID number?

A: Each endpoint of a state should have a different trace event ID number. Usually each trace event logging routine logs a different trace event ID number. This lets you easily identify which source line logged the trace event, how often that source line executed, and what order source lines executed in. However, it is sometimes useful to log the same trace event ID in multiple places. This makes it possible to group trace events from related, but not identical, activities. For more information, see "trace_event and Its Variants" on page 2-12.

Q: How can I prevent user trace events from being discarded or lost?

A: Use expansive mode; avoid use of buffer or file wrapping options. Flush the shared memory buffer more often by tuning:

- The shared memory buffer 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 5-1 and Chapter 3 for more information.

Q: What can I do if trace events are not appearing in an ntrace display?

A: Press Refresh, fill out the Search Form, fill in values in the interval control area, use the interval scroll bar, keep pressing the Zoom Out 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.

Q: How can I prevent kernel trace events from being lost?

A:

- 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 5-1 and Chapter 3 for more information.

Q: Why can't I see my individual thread names?

A: By default, all threads will share the same thread name (either "main" or the thread name passed to trace_open_thread()). You can specify a new thread name in individual threads using trace_open_thread after registering your thread with the NightTrace API. See "Threads and Logging" on page 2-26 for more information.

F
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 process, *Ada task*, or *thread* (or a set of any combination of these) that is associated with a uniquely named *trace context*. The thread name is derived from the argument specified to the `trace_open_thread()` function.

NightTrace thread identifier

See *TID*.

NightView

A symbolic debugger that is part of the NightStar tool kit. It lets you debug C and Fortran applications; these applications may be composed of one or more processes, running on one or more CPUs. Among other things, NightView can automatically patch trace event logging routines into your executable application.

node

A system from which a *trace event file* can come from.

node box

If the RCIM synchronized tick clock is used to timestamp events, this is a *Grid Label* on a kernel *display page*. It identifies which *node* to which the displayed data corresponds.

node ID

A unique identifier internally assigned by NightTrace to every *node* that has an *trace event file* in a trace file analysis.

node name

The name of a system from which a *trace event file* can come.

node_name table

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and associates *node ID* numbers with *node names*.

node PID table

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and associates process identifiers (*PIDs*) with process names for a particular *node*. The name of each node's table is `pid_`*nodename* where *nodename* is the node's name. If kernel tracing, this table is stored in the **`vectors`** file.

node TID table

A pre-defined, dynamically generated *string table*. It is internal to NightTrace. If user tracing, it associates NightTrace thread ID numbers with thread names for a particular *node*. If kernel tracing, this table is not used. The name of each node's table is `tid_`*nodename* where *nodename* is the node's name.

NT_ASSOC_PID

An overhead *trace event* that **`ntraceud`** logs at the beginning and end of each process.

NT_ASSOC_TID

An overhead *trace event* that **`ntraceud`** logs at the beginning and end of each *thread* and *Ada task*.

NT_CONTINUE

An overhead *trace event* that **`ntraceud`** logs for multi-argument trace events.

ntrace display utility

The part of *NightTrace* that graphically displays *trace events*, trace event data, and *states* for debugging and performance analysis.

ntraceud

The *NightTrace* daemon process that allows you to log user-defined *trace events* and data from user applications written in C, 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 context

All *trace points* are associated with a log file (established via `trace_start`) and a thread name (established via `trace_open_thread`). If two processes (or *tasks* or *threads*) are associated with the same log file and thread name, then they are said to have the same trace context. If they differ in log file, thread name, or both, then they have different trace contexts.

trace event

A user-defined point of interest in an application's source code that NightTrace represents with an integer *trace event ID*. Alternatively this may be a predefined point of interest in the kernel. Along with the trace event ID, *NightTrace* records the *timestamp* when the trace event occurred, any arguments logged with the trace event, and the logging process identifier (*PID*).

trace event argument

A user-defined numeric value logged by an application via a *trace event*.

trace event file

An **`ntraceud`**-created binary file that contains sequences of *trace events* and data that your application and the **`ntraceud`** daemon logged.

trace event function

The class of NightTrace *functions* that provide information about *trace events*. They operate on either the *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 Night-Trace display utility consists of several windows.

window manager

The program that controls *window* placement, size, and operations.

wraparound mode

The mode that causes the **`ntraceud`** daemon to intentionally discard old events. There are two forms of wraparound mode: *buffer-wraparound* and *file-wraparound*. The other **`ntraceud`** mode is *expansive mode*.

Index

G

N

O

P

Q

R

S

T

V

W

X

Z