NightTrace Manual

CONCURRENT COMPUTER CORPORATION™

0890398-070

August 2000

Printed in U. S. A.

Revision History:	Level:	Effective With:
Original Release -- August 1992	000	NightTrace 1.0
Current Release -- August 2000	070	NightTrace 4.2

Preface

Scope of Manual

This manual is a reference document and users guide for NightTrace™[1], a graphical, interactive debugging and performance analysis tool, and KernelTrace™, a tool that collects and textually analyzes system performance.

Structure of Manual

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

- Chapter 1 contains introductory material on NightTrace and KernelTrace.
- Chapter 2 describes system and user environmental requirements.
- Chapter 3 gives the syntax and examples of NightTrace library calls.
- Chapter 4 tells how to generate trace event logs with **`ntraceud`**.
- Chapter 5 describes how to invoke the **`ntrace`** display utility.
- Chapter 6 shows how to view trace event logs with **`ntrace`**.
- Chapter 7 illustrates **`ntrace`** display objects and their creation.
- Chapter 8 shows how to configure **`ntrace`** display objects.
- Chapter 9 defines NightTrace expressions.
- Chapter 10 tells about NightTrace's built-in tools.
- Chapter 11 describes kernel tracing with **`ktrace`**, **`ntfilter`**, and **`ntrace`**.

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

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

The glossary contains an alphabetical list of NightTrace, X™[2], and Motif™[3] words and phrases used in this manual and their definitions. The index contains an alphabetical list of topics, names, etc. found in the manual.

1. NightTrace is a trademark of Concurrent Computer Corporation
2. X Window System and X are trademarks of The Open Group
3. Motif, OSF, and OSF/Motif are trademarks of Open Software Foundation, Inc.

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.
➩	An arrow separates a menu bar item from a pull-down menu entry.

The window images in this manual come from a Motif environment. If you are using another environment, your windows may differ slightly from those presented here.

Referenced Publications

The following publications are referenced in this document:

0830048	HN6200 Architecture Reference Manual
0830046	HN6800 Architecture Manual
0890240	hf77 Fortran Reference Manual
0890300	X Window® System User's Guide
0890378	C: A Reference Manual
0890380	OSF/Motif™ Documentation Set (3 volumes)
0890395	NightView™[4] User's Guide
0890423	PowerMAX OS™[5] Programming Guide

0890429	System Administration Volume 1
0890430	System Administration Volume 2
0890460	Compilation Systems Volume 2 (Concepts)
0890466	PowerMAX OS™ Real-Time Guide
0890474	NightTrace™ Pocket Reference
0890516	MAXAda™ Reference Manual
0891019	Concurrent C Reference Manual
0891055	Élan™[6] License Manager Release Notes
0891082	Real-Time Clock and Interrupt Module User's Guide
	PowerPC™[7] 604 RISC Microprocessor User's Manual (not available from Concurrent Computer Corporation)

4. NightView is a trademark of Concurrent Computer Corporation
5. PowerMAX OS is a trademark of Concurrent Computer Corporation
6. Élan License Manager is a trademark of Elan Computer Group, Inc.
7. PowerPC is a trademark of International Business Machines, Corp.

Contents

1 Introduction

1 Introduction

Overview

This chapter provides an overview of NightTrace and KernelTrace, steps involved in using both tools, and recommended readings.

What is NightTrace?

NightTrace is an interactive debugging and performance analysis tool, a part of the Night-Star™ tool kit. NightTrace allows you to graphically display information about important events in your application and the kernel, for example, event occurrences, timings, and data values. NightTrace consists of the following parts:

NightTrace library	Routines in user applications that log trace events to shared memory
`ntraceud`	Daemon process that copies user applications' trace events from shared memory to trace event file(s)
`ntrace`	Tool that graphically displays user and kernel trace events in trace event file(s)

The KernelTrace tool set allows you to collect and textually analyze kernel trace information. It also allows you to convert kernel trace information into NightTrace format for graphical analysis. KernelTrace consists of the following parts:

`ktrace`	Tool that collects and textually analyzes kernel trace events
`ntfilter`	Tool that converts KernelTrace trace event files from **`ktrace`** into NightTrace trace files that **`ntrace`** can display

NightTrace and KernelTrace are flexible. As a user, you control:

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

- The amounts and types of information you display
- Trace event searches and summaries

User and Kernel Tracing

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

Timestamp Source Selection

By default, the interval timer (NightHawk 6000 Series) or the Time Base Register (PowerHawk/PowerStack) is used to timestamp trace events. However, NightTrace and KernelTrace can specify the Real-Time Clock and Interrupt Module (RCIM) as a timestamp source.

The RCIM is an optional hardware module, attached to a single-board computer (SBC), which contains a tick clock that can be synchronized between several SBCs by way of an interconnection cable. This synchronized tick clock can be used as a common time base for both kernel-level tracing and user-level tracing across multiple SBCs. NightTrace supports using the RCIM synchronized tick clock to timestamp trace events and also supports displaying trace data generated on multiple SBCs having the common time base. The RCIM also contains a POSIX clock. However, the POSIX clock is not supported as a timestamp source by NightTrace.

Selection of the RCIM synchronized tick clock as the trace timing source is made via the **`-clock rcim_tick`** option to both **`ntraceud`** and **`ktrace`** .

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

Trace-Point Placement

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

Some typical user trace-point locations include:

- Suspected bug locations
- Process, subprogram, or loop entry and exit points
- Timing points, especially for clocking I/O processing
- Synchronization points/multi-process interaction

- Endpoints of atomic operations
- Endpoints of shared memory access code

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

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

Languages Supported

The NightTrace library is callable from C, Fortran and Ada. This means that your application can be written in any combination of these languages and still log trace events.

Processes and CPUs

`ntraceud` (the ntrace user daemon) is responsible for actually recording the trace events logged by an application to disk. It can interact with single-process and multi-process applications; the processes may even run on different CPUs. When you log a trace event, NightTrace identifies both the originating process and optionally the CPU.

Information Displayed

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

Searches and Summaries

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

Logging and Analysis

NightTrace and KernelTrace support two activities: trace event logging and trace event analysis.

The Trace Event Logging Procedure

The following text describes user and kernel trace event logging. If you are interested only in kernel tracing, skip the steps that are specific to user tracing. If you are interested only in user tracing, skip the steps that are specific to kernel tracing. For trace event logging, follow these steps in the order shown:

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

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

main()
{
   trace_start( "log" );
   trace_open_thread( "main_thread" );
   trace_event( START );

   process();

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

Figure 1-1. Example of Instrumented C Code

5. (For user traces only) Compile and link your application with the Night-Trace trace event logging library. For example:

```
$ cc main.c process.c -lntrace -lud
```

6. (For RCIM synchronized tick clock only) Synchronize the tick clocks on all connected RCIMs before kernel and/or user tracing has begun.

   Use the **clock_synchronize(1M)** command.

7. (For kernel traces only) Invoke the **ktrace** tool in the background. This permits you to log kernel trace events simultaneously with user trace events. For example:

   ```
   $ ktrace -o raw_klog &
   [1] 452
   locking into memory
   resetting priority
   open /dev/trace
   initialize
   gather trace point data
   ```

   Note that if you are running this command from a script, you may need to sleep for about 5 seconds so the "gather trace point data" message has time to appear.

NOTE

> In order to use the RCIM synchronized tick clock, you will need to specify the **-clock rcim_tick** command line option when invoking **ktrace**. For more information, please see the **ktrace(1)** man page.

8. (For user traces only) Invoke the **ntraceud** daemon, so it can log trace events for your application. For example:

   ```
   $ ntraceud log
   ```

   (Note that the trace event file on the call to the `trace_start` library routine (step 4) matches the trace event file on the **ntraceud** invocation (step 7).)

NOTE

> In order to use the RCIM synchronized tick clock, you will need to specify the **-clock rcim_tick** command line option when invoking **ntraceud**. For more information, please see the **ntraceud(1)** man page.

9. (For user traces only) Run your application. As NightTrace trace event logging routines execute, they write trace event information into a shared memory buffer. Periodically, the **ntraceud** daemon copies this information to a trace event file on disk. For example:

   ```
   $ a.out
   ```

10. (For user traces only) When the application completes, stop the **ntraceud** daemon. For example:

    ```
    $ ntraceud -quit log
    ```

11. (For kernel traces only) Kill the **ktrace** tool, so it stops logging kernel trace events. For example:

    ```
    $ kill %1
    terminating
    ```

12. (For kernel traces only) Invoke the **ntfilter** tool to convert the Kernel-Trace trace event file (**raw_klog**) that the **ktrace** tool created into one compatible with NightTrace (**klog**). For example:

    ```
    $ ntfilter -v < raw_klog > klog
    ```

13. (For user traces only) Create an event-map file for **ntrace** (optional). An event-map file provides a mechanism for associating meaningful symbolic tags with the different trace event IDs logged by the application. When **ntrace** reads an event-map file, it can display the symbolic tags for trace events; otherwise, it must display the more cryptic numeric trace event IDs. For example (Assume the event-map file name is **map**.):

    ```
        event: 10 “START” 1 %d
        event: 20 “END” 1 %d
    ```

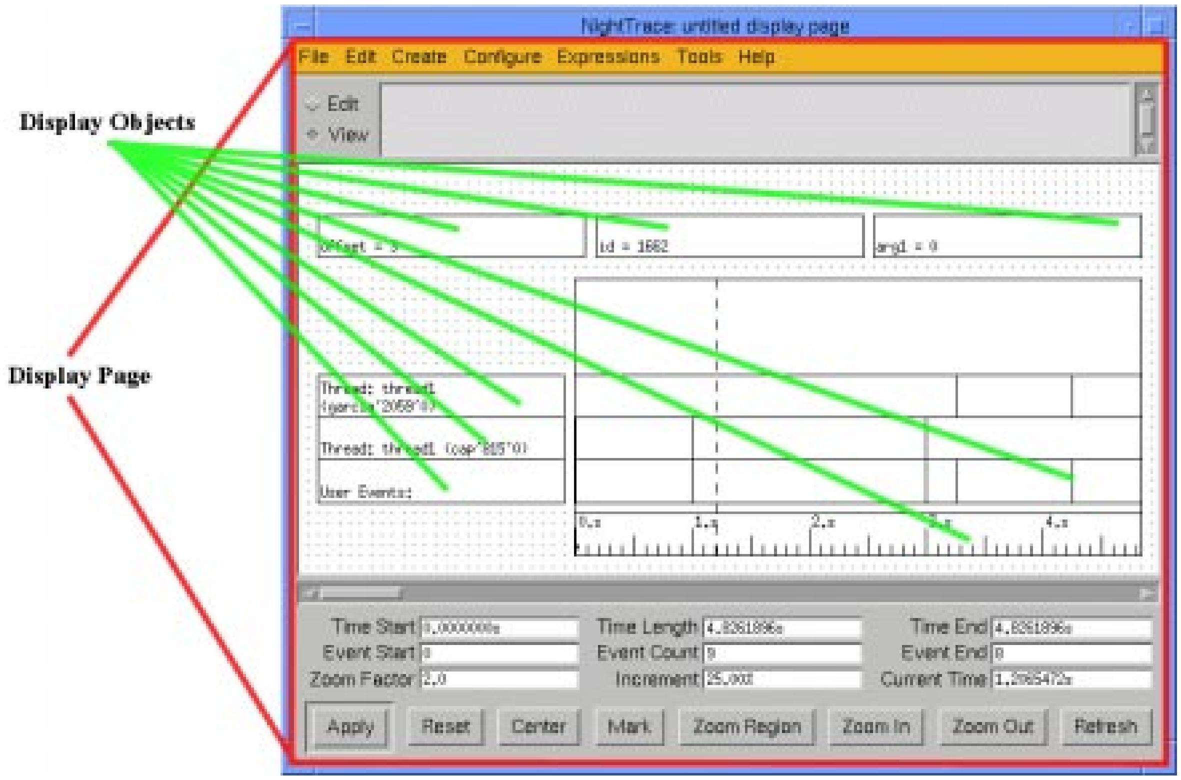
The Trace Event Analysis Procedure

When trace event logging completes, you can begin trace event analysis. For graphical trace event analysis, follow these steps in the order shown:

1. Invoke the **ntrace** display utility. Command-line options limit which trace events are loaded. Command-line arguments usually include user trace event file(s) and possibly a kernel trace event file, **vectors** file, and user-created files that customize and annotate your displays. For example:

    ```
    $ ntrace log map klog vectors
    ```

2. Create or modify *display pages*. Display pages contain built-in graphical components called *display objects*. There are pre-defined display pages for user and kernel traces. You select and configure display objects so they meet your needs. This usually means graphically displaying chronological trace event or state information that your application and/or the kernel logged. When you finish customizing your display pages, save them for future use. Figure 1-2 shows an example of a user display page.



Time Start 0.0000000s
Event Start 0
Zoom Factor 2.0
Time Length 4.8261896s
Event Count 9
Increment 25.000
Time End 4.8261896s
Event End 8
Current Time 1.2065472s

Figure 1-2. Example of a User Display Page with Display Objects

3. Iteratively locate and analyze significant data.

   • Search for trace events of interest. You do this by controlling the window that displays a portion of the trace event file. This window is called the *interval*. You can control the interval by zooming in or out, scrolling, searching for specific trace events, or jumping to portions of the trace event file.

   • Display summary information. This information may be about your entire trace session or the characteristics of particular trace events and states in this trace session.

For textual kernel trace event analysis, follow the step shown:

```
$ ktrace -input rawfile > summary_file
```

Recommended Reading

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

X Window System User's Guide

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

OSF/Motif Style Guide

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

2
Establishing the Environment

2
Establishing the Environment

Overview

This chapter describes the system and user environment you must have before you can run NightTrace and KernelTrace.

Requirements

NightTrace and KernelTrace require a particular system and user environment in which to run. It is your system administrator's responsibility to establish this environment. Some tasks that must be performed include:

- Install the software
- Configure the kernel
- Administer privileges
- Put NightTrace users into groups based on their needs (optional):
    - Users that need page lock privilege (`P_PLOCK`)
    - Users that need access to the system's interrupt priority level (IPL) register

      (Access to the IPL register and page lock access reduce trace event logging overhead for time-critical applications.)
- Grant read access to the kernel trace device to all KernelTrace users
- Grant read access to timestamp source
    - Interval Timer

      Grant read access to the system's interval timer to all NightTrace and KernelTrace users
    - RCIM Synchronized Tick Clock

      Grant read access to the RCIM synchronized tick clock to all NightTrace and KernelTrace users

NOTE

Granting read access to the timestamp source is not necessary when using the Time Base Register on Power Hawk systems.

Installing Software

All NightStar tools, including NightTrace's **`ntrace`** program, must be run with the Élan License Manager. The `elanlm` package contains files for the Élan License Manager. Follow the steps in the "Obtaining Licenses" section of the *Élan License Manager Release Notes*; the *feature alias* is `NightTrace`. If you are not already running the Élan License Manager, if you do not have a copy of the *Élan License Manager Release Notes*, or if you need a license key, contact Concurrent Software Distribution at 1-800-666-5405.

NOTE

If your system is already running the Élan License Manager, you may not need to reinstall it.

The `ntrace` package contains files for NightTrace, and the `trace` package contains files for KernelTrace. The following example installs the NightTrace `ntrace` package from a tape device named `tape1`:

```
pkgadd -d tape1 ntrace
```

For more information, see **`pkgadd(1M)`**, *PowerMAX OS Version 2.1 Release Notes*, and "Installing Add-On Software" in *System Administration Manual Volume 1*.

NOTE

To determine where NightTrace will be installed on your system, look at the setting of the `basedir` parameter in the **`/var/sadm/install/admin/default`** file. (For possible values, see **`admin(4)`**.)

Configuring the Kernel

Table 2-1 describes the kernel tunable parameter that affects KernelTrace.

Table 2-1. Significant Kernel Tunable Parameter

Kernel Tunable Parameter	Description
`TR_BUFFER_COUNT`	Number of kernel trace buffers (For more information, see “Kernel Tracing with ktrace” on page 11-8.)

Table 2-2 describes the kernel options that NightTrace and KernelTrace require.

Table 2-2. Required Kernel Options

Kernel Option	Description
`fp`	Fixed-priority class scheduler
`ipc`	Inter-process communications. (NightTrace applications log trace events to a shared memory buffer. For more information about shared memory, see “Interprocess Communication” in the *PowerMAX OS Programming Guide*.)
`procfs`	Processor file system
`trace`	Kernel trace driver.

Refer to “Booting and System States” in *System Administration Volume 1*, “Configuring and Building the Kernel” and “Tunable Parameters” in *System Administration Volume 2*, **`idbuild(1M)`** and **`idtune(1M)`** for instructions on modifying kernel configurations.

Administering Privileges

NightTrace and KernelTrace use sensitive real-time system services that require special privileges that are not generally available to all users and processes. Table 2-3 shows the privileges that processes must have to run NightTrace and KernelTrace.

Table 2-3. Required Privileges

P_OWNER	P_FPRI	P_USERINT
P_DEV	P_TSHAR	
P_SYSOPS	P_PLOCK	

Privileges are associated with users, executable files on disk, and executing processes. However, ultimately, the set of privileges associated with an executing process is most important.

NOTE

If a system service call requires a specific privilege, any process calling that system service must also have that privilege.

The granting of privileges to users, executable files and processes is a complex issue and depends on the specific security configuration of each system. For a complete description of privileges and security refer to the “Trusted Facility Management” Chapter in *System Administration Volume 1* and **`intro(2)`**.

A convenient way to associate privileges with users is through the use of *roles*. A role is a named description of a set of privileges that have been registered for certain executable files, such as the shell. The system administrator creates roles and assigns users to them. During the login process, use the **`tfadmin(1M)`** command to request that your shell be granted the privileges associated with your role. Once privileges have been granted to your shell, subsequently spawned processes automatically inherit your privileges.

TIP:

The system administrator should issue the following commands to create a role and register all the privileges that NightTrace programs require to three commonly used shells (**`sh`**, **`ksh`**, and **`csh`**).

```
$ /usr/bin/adminrole -n TRACE_USERS

$ /usr/bin/adminrole -a sh:/sbin/sh:owner:dev: \
    sysops:fpri:tshar:plock:userint TRACE_USERS

$ /usr/bin/adminrole -a ksh:/usr/bin/ksh:owner:dev:\
    sysops:fpri:tshar:plock:userint TRACE_USERS

$ /usr/bin/adminrole -a csh:/usr/bin/csh:owner:dev: \
    sysops:fpri:tshar:plock:userint TRACE_USERS
```

TIP:

The system administrator should issue the following command to assign an example user (*Kernel_Jock*) to the TRACE_USERS role.

```
$ /usr/bin/adminuser -n -o TRACE_USERS Kernel_Jock
```

Kernel_Jock must explicitly request privileges for the current shell by initiating a new shell with the **`tfadmin(1)`** command.

TIP:

For convenience, *Kernel_Jock* should put the following line in his **`.profile`** (or **`.login`**) file. (This file is executed during initialization of the login shell.)

```
exec /sbin/tfadmin TRACE_USERS: $SHELL
```

This causes the privileges associated with the TRACE_USERS role to be automatically granted to a newly spawned shell (`SHELL` is an environment variable that is automatically set by the shell; it represents *Kernel_Jock*'s actual default shell path name; e.g., **`/usr/bin/ksh`**). The original shell that executed the **`.profile`** (or **`.login`**) file is replaced by the new shell spawned by the **`tfadmin`** command.

Kernel_Jock can now run NightTrace and KernelTrace.

Putting Users into Groups

It is possible for one user to belong to several groups in the **`/etc/group`** file. Proper group assignment permits limited use of restricted-access resources to users who really need them.

TIP:

Your system administrator should consider putting NightTrace and KernelTrace users into three groups in the **`/etc/group`** file. There could be one group for each of the following:

- All NightTrace and KernelTrace users. Everyone in this group must be able to read the system's interval timer (**`/dev/interval_timer`**).
- Those users whose applications must not be rescheduled or interrupted., for example, those who are using user-level interrupts. Everyone in this group must be able to read and write to the interrupt priority level register (**`/dev/spl`**).
- Those users who perform kernel traces. Everyone in this group must be able to read the kernel trace device (**`/dev/trace`**).

For more information about adding groups and users to the system, see **`useradd(1M)`**, **`usermod(1M)`**, and **`groupadd(1M)`**.

Granting Page Lock Privilege

NightTrace does not require you to have page lock (`P_PLOCK`) privilege. However, if you have it, you can prevent page faults within the NightTrace trace event logging routines, and optionally within your application. Page faults can distort your trace event timings and can degrade the efficiency of applications and facilities.

Usually, users are denied `P_PLOCK` privilege. By default, the **`ntraceud`** daemon and the library initialization routine use page locking.

From an `P_PLOCK`-privilege standpoint, NightTrace users fall into two categories:

- Those who have `P_PLOCK` privilege can prevent paging
- Those who lack `P_PLOCK` privilege must accept paging

The following sections describe how NightTrace performs with and without the privilege to lock pages in memory.

Using Page Locking

This section discusses the following `P_PLOCK` privilege issues:

- What applications require it
- What are the advantages of it
- Why does NightTrace use it
- What action must your system administrator take

Applications that typically lock pages in memory include the following: user applications that log trace events and, by default, the **`ntraceud`** daemon. These applications must be able to lock their pages in memory. Note: the NightTrace library routines lock only their critical code and data pages in memory; you need not lock the entire application.

By locking pages in memory, **`ntraceud`** and the NightTrace library routines in user applications prevent page faults during traces. Otherwise, this overhead can distort trace event timings.

The NightTrace library uses page locking for two reasons. First, its routines need to synchronize themselves when they are used at program level and in user-level interrupt routines. The system cannot afford the overhead of a page fault in a NightTrace library routine while a user-level interrupt is waiting for the routine to complete. Second, the NightTrace library routines must be very efficient to reduce any performance and timing impact on the user application.

To keep your applications from being paged out of memory, your system administrator must grant you `P_PLOCK` privilege. You can query your privileges with the **`priv`** special command of **`/sbin/sh`**. The system administrator can set privileges with the **`adminuser(1M)`** command.

Not Using Page Locking

This section discusses the lack of `P_PLOCK` privilege as it applies to these topics:

- What can you do if you lack it
- What are the disadvantages of lacking it

If you lack `P_PLOCK` privilege, you must invoke the **`ntraceud`** daemon with the **`-lockdisable`** option. This option makes **`ntraceud`** and the NightTrace library routines in your application run without locking their pages in memory.

With this option you are able to log trace events. However, the overhead of the trace event logging routines may increase due to paging, exceptions and interrupts.

NOTE

`ntraceud` always protects the data integrity of its shared memory buffer with spin locks. If a page fault occurs while this spin lock is locked, all other processes contending for the spin lock wait until the page-faulted application is paged in and rescheduled and logging of the trace event is completed. Locking the NightTrace library in memory assures that the application will not page fault while logging a trace event to the shared memory buffer.

For more information on `P_PLOCK`, see **`intro(2)`**. For more information on **`ntraceud`** options, see “ntraceud Options” on page 4-4 and “Option to Prevent Page Locking (-lockdisable)” on page 4-9.

Granting Access to the Interrupt Priority Level Register

NightTrace does <u>not</u> require you to read and write the system’s interrupt priority level register (IPL). However, if you can modify this register, you can prevent rescheduling and interrupts during trace event logging; they can distort trace event timings and can degrade the efficiency of applications and facilities.

Usually, users are denied IPL modification access because it means relaxing system protection that normally limits IPL modification to the operating system. By default, the **`ntraceud`** daemon and library initialization routine modify the IPL register.

From an IPL-modification standpoint, NightTrace users fall into two categories:

- Those who have IPL modification access and can prevent rescheduling and interrupts
- Those who lack IPL modification access and must accept rescheduling and interrupts

The following sections describe how NightTrace performs with and without access to modify the IPL register.

Using the IPL Register

This section discusses the following IPL modification access issues:

- What applications require it
- What are the advantages of it
- How does NightTrace use it
- What action must your system administrator take

Applications that typically modify the system's IPL register to prevent rescheduling and interrupts include the following: user applications that log trace events and, by default, the **`ntraceud`** daemon. These programs must be able to read and write to the system's IPL register.

By modifying this normally restricted system register, **`ntraceud`** prevents rescheduling and interrupts during traces. Otherwise, this overhead could distort trace event timings.

Applications that can modify the IPL register, temporarily raise their own priority in the system's IPL register. This way they prevent rescheduling and interrupts during trace event logging. NightTrace then locks a spin lock on the shared memory buffer. This protects shared memory. Once logging of the trace event is complete, **`ntraceud`** unlocks the spin lock and lowers the IPL register value back to zero.

By default, the NightTrace library routines open **`/dev/spl`** on Series 6000 systems or **`/dev/spl1`** and **`/dev/spl2`** on Power Hawk systems to gain access to the system's IPL register. User applications do not explicitly access the system's IPL register through NightTrace library routines.

If an application lacks the read and write access to these device files, the **`ntraceud`** daemon and library initialization routine exit with errors. If errors are detected, your system administrator must do at least one of the following:

- Add you to the user group who has **`/dev/spl`** read and write permissions (or **`/dev/spl1`** and **`/dev/spl2`** for Power Hawk systems)
- Grant read and write access to IPL register users

Not Using the IPL Register

If you lack read or write access to the system's IPL register, you must invoke the **`ntraceud`** daemon with the **`-ipldisable`** option. This option prevents **`ntraceud`** and the NightTrace library routines in your application from modifying the system's IPL register.

With this option, you are able to trace events. However, their timings may be distorted due to process rescheduling and interrupts.

The **`-ipldisable`** option should be used with great care. Using it may lead to deadlock if more than one LWP, each biased to run on the same CPU, is logging trace events to a trace file created by an **`ntraceud`** invoked with this option. Consider the following scenario: an LWP, preparing to log a trace event, locks the spin lock to protect the shared memory buffer. It is preempted (through a rescheduling interrupt) by a second LWP which also attempts to log a trace event. However, due to priority inversion, the first LWP cannot release the spin lock, causing the second LWP to loop infinitely waiting for the spin lock to be released.

NOTE

`ntraceud` always protects the data integrity of its shared memory buffer with spin locks. If rescheduling or an interrupt occurs while this spin lock is locked, all other processes contending for the spin lock wait until the preempted process is rescheduled and logging of the trace event is completed. Using the IPL register and locking the NightTrace library pages in memory prevents this.

For more information on the system's IPL register, see "User-Level Interrupts" in the *PowerMAX OS Real-Time Guide*. For more information about spin locks, see "Interprocess Synchronization" in the *PowerMAX OS Real-Time Guide*. For more information on **`ntraceud`** options, see "ntraceud Options" on page 4-4 and "Option to Disable the IPL Register (-ipldisable)" on page 4-8.

Granting Access to the Trace Device

The **`ktrace`** kernel-trace tool requires that its users have read access to **`/dev/trace`**, the kernel trace device. This access is not required for user tracing.

Granting Access to the Interval Timer

NOTE

This section does not apply to Power Hawk systems. On those systems, the time base register of the microprocessor is used for trace event timestamps.

The NightTrace library routines open **`/dev/interval_timer`** to gain access to the system's interval timer These routines in your application use this timer when they write trace events to a shared memory buffer.

Although user applications must be able to read the interval timer, they do not explicitly access it. (The NightTrace event-logging library accesses it.) Usually, system users are unable to access the interval timer because it means relaxing system protection that normally limits interval timer access to the operating system.

If applications lack read access to **`/dev/interval_timer`**, the NightTrace daemon and library initialization routine exit with errors. If errors are detected, your system administrator must do at least one of the following:

- Add you to the user group that has **`/dev/interval_timer`** read permission
- Grant read access to interval timer users

CAUTION

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

Granting Access to the RCIM Synchronized Tick Clock

NOTE

This section applies only to those systems on which an RCIM is installed, configured and functioning.

When trace events are to be timestamped by the RCIM tick clock, the NightTrace library routines open **`/dev/sync_clock`** to gain access to the clock.

Although user applications must be able to read the RCIM synchronized tick clock, they do not explicitly access it. (The NightTrace event-logging library accesses it.)

If applications lack read access to **`/dev/sync_clock`**, the NightTrace daemon and library initialization routine exit with errors. If errors are detected, your system administrator must do at least one of the following:

- Add you to the user group that has **`/dev/sync_clock`** read permission
- Grant read access to the RCIM tick clock

CAUTION

On Power Hawk and Power Stack systems, do not start, stop, or synchronize the RCIM synchronized tick clock in the middle of gathering trace events. Any one of these acts will render trace data useless because it interferes with obtaining a valid timestamp. Also, there are certain situations in which the RCIM clock values may be synchronized without direct user intervention. Again, any one of these occurrences might invalidate trace data. For more information, please see the *Real-Time Clock and Interrupt Module User's Guide*.

3 Adding Library Calls to Your Application

3
Adding Library Calls to Your Application

Overview

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

CAUTION

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

CAUTION

The NightTrace Version 4.1 **`ntraceud`** is incompatible with user programs statically linked with NightTrace libraries prior to Version 4.1. This is due to a change in the layout of the shared memory region used to provide communication between **`ntraceud`** and user programs.

Any user programs statically linked with these libraries will need to be relinked with the Version 4.1 libraries. Failing to relink the application with the new libraries can result in unpredictable behavior or in the application looping infinitely when it calls `trace_open_thread()`.

Beginning with the NightTrace 4.1 static library, applications can detect when they are not compatible with **`ntraceud`** and will exit with an error code instead of exhibiting undesired behavior. Programs linked with an earlier version of the static library cannot detect this incompatibility.

Language-Specific Source Considerations

NightTrace applications must be written in C, Fortran, or Ada. For your applications to trace events, you must edit your source code and insert NightTrace library routine calls (unless you are using the NightView debugger). This is called *instrumenting your code*. Before you begin this task, you should read the appropriate language section below.

C

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

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

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

- Function prototypes for all NightTrace library routines
- Return values for all NightTrace library routines
- C macros (described in “Disabling Tracing” on page 3-24)

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

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

For more information on C, see *C: A Reference Manual* and the Concurrent *C Reference Manual*.

Fortran

The Fortran version of the NightTrace library routines follow **`hf77`** function-naming and argument-passing conventions. For more information on **`hf77`**, see the *hf77 Fortran Reference Manual*.

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

```
integer trace_start
```

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 Reference Manual.*

Inter-Process Communication and Library Routines

Your application logs trace events to the shared memory buffer. Later, the **`ntraceud`** daemon copies trace events from the shared memory buffer to the trace event file. The relationship between your application and the **`ntraceud`** daemon and the sequence of library calls needed to maintain this relationship appears in Figure 3-1.

Understanding NightTrace Library Calls

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

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

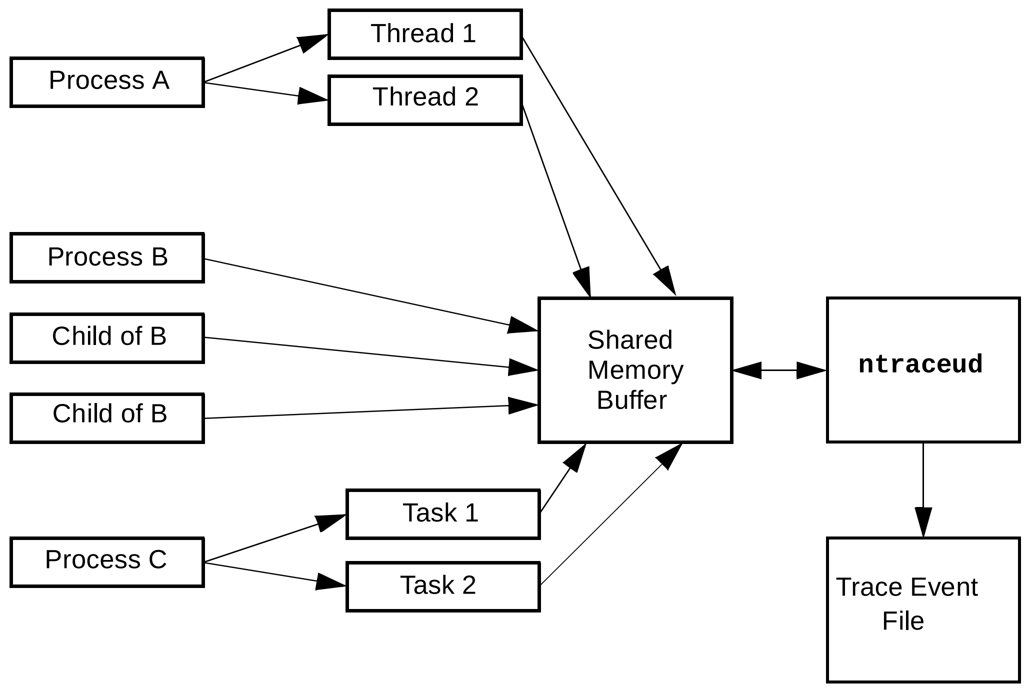
See the *NightTrace Pocket Reference* card for a syntax summary of these routines. The next sections describe these routines in detail.

Parent processes follow this sequence:

- trace_start()
- trace_open_thread()
- log trace events
- trace_close_thread()
- trace_end()

Child processes follow this sequence:

- trace_open_thread()
- log trace events
- trace_close_thread()



An application written in C can log trace events using:

- trace_event()
- trace_event_arg()
- trace_event_flt()
- trace_event_two_flt()
- trace_event_dbl()
- trace_event_two_dbl()
- trace_event_four_arg()

and it can control which trace events are logged and when they are written to disk using:

- trace_enable()
- trace_enable_range()
- trace_enable_all()
- trace_disable()
- trace_disable_range()
- trace_disable_all()
- trace_flush()
- trace_trigger()

Figure 3-1. Inter-Process Communication and Library Routines

trace_start()

The `trace_start()` routine initializes the trace mechanism and acquires resources for your process, C thread or Ada task.

SYNTAX

```
C:          int trace_start(char *trace_file);

Fortran:    integer function trace_start(trace_file)
        character *(*) trace_file

Ada:        function trace_start(trace_file : string)
        return ntrace_error;
```

PARAMETERS

trace_file	**`ntraceud`** logs trace events to an output file, *trace_file*. When you invoke the **`ntraceud`** daemon, you must specify this file's name. For **`ntraceud`** to log your process' trace events to this file, the trace event file parameter in your `trace_start()` call must correspond to the trace event file argument on the **`ntraceud`** invocation line. This means that the names do <u>not</u> have to match exactly, but they do have to refer to the same inode; for example, one path name may begin at your current working directory and the other may begin at the root directory.

DESCRIPTION

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

- Verifies that an instance of **`ntraceud`** is running with the matching trace event file name
- Verifies that the version of the NightTrace library linked with the application is compatible with the version being used by **`ntraceud`**
- Verifies that the RCIM synchronized tick clock is counting if it was selected as the timestamp source
- Attaches the **`ntraceud`**-created shared memory buffer
- Attaches the shared memory regions bound to the timestamp source and interrupt priority level (IPL) register
- Locks critical NightTrace library routine pages in memory
- Initializes trace event tracing in this process

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

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

Generally, call `trace_start()` only once per parent process. However, for processes using C threads or Ada tasks, `trace_start()` can be called by individual threads or tasks, allowing a specific thread or task to log trace events to a unique trace event file. For detailed guidelines on `trace_start()` placement, see Figure 3-2.

For processes using C threads and Ada tasks, all threads and tasks will inherit the trace context of the first `trace_start()` call that is made by any thread or task of the process. However, subsequent `trace_start()`calls by a thread or task will override the default trace context. Newly created threads and tasks always inherit the trace context of the thread or task that created them.

RETURN VALUES

The `trace_start()` routine returns a zero value (`NTNOERROR`) on successful completion. Otherwise, it returns a non-zero value to identify the error condition. If `trace_start()` returns any error code other than `NTALREADY`, the application cannot do a trace. A list of `trace_start()` error codes follows.

`[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_start()` 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 **`ntraceud`** daemon. Solution: Relink the application with the static library version which matches the library version being used by the daemon. Note: This error code will be returned only if the application is linked with the version 4.1 or later static library. Applications linked with a static library version previous to 4.1 must be relinked to take advantage of this compatibility check.
`[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, restart **`ntraceud`**, and restart the application.
`[NTNOTRACEFILE]`	The trace event file does not already exist. This often means that **`ntraceud`** is not running. Solution: Be sure that an **`ntraceud`** daemon is running with the same trace event file name as the `trace_start()` parameter.

`[NTNODAEMON]`	The trace event file exists, but no shared memory region exists. This usually means that there is no **`ntraceud`** daemon running with a trace event file name that matches the one on the `trace_start()` call. Solution: If the **`ntraceud`** daemon is not running, invoke it. If the file names do not match, either invoke **`ntraceud`** with the correct file name or edit your source code. This can also occur if the shared memory region exists, but there is no evidence of a **`ntraceud`** daemon currently running (e.g., it aborted abnormally). This condition is not always detectable. Solution: Remove the shared memory region with **`ipcrm(1)`** and restart **`ntraceud`**.
`[NTPERMISSION]`	The calling application lacks permission to attach the shared memory buffer. Solution: Make sure that the same user who started up **`ntraceud`** is the current user logging trace events in the application.
`[NTMAPSPLREG]`	The system's IPL register could not be attached. Solution: If read or write access is lacking, see your system administrator or invoke **`ntraceud`** with the **`-ipldisable`** option.
`[NTPGLOCK]`	Permission to lock the text and data pages of the NightTrace library routines was denied. Solution: If `P_PLOCK` privilege is lacking, see your system administrator or invoke **`ntraceud`** with the **`-lockdisable`** option.

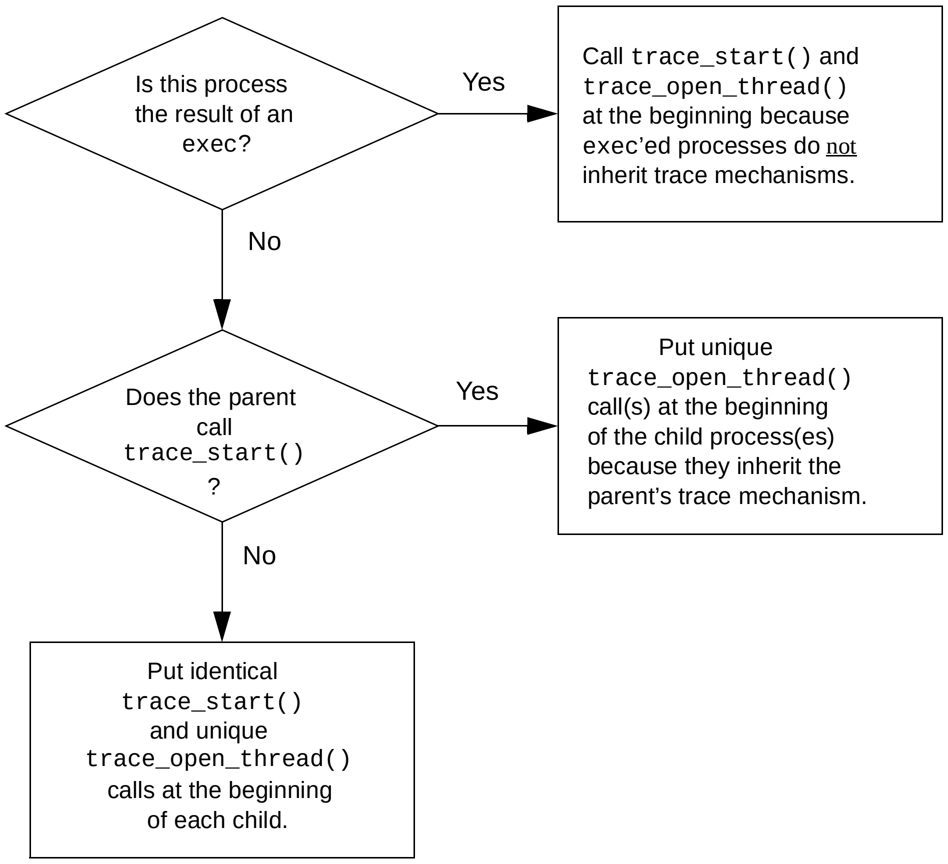
SEE ALSO

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

See "ntraceud Options" on page 4-4 for more information on **`ntraceud`** options.

Several conditions in the application warrant `trace_start()` and `trace_open_thread()` calls. These situations appear in the flowchart below.

Note: All these cases assume that you want to do tracing in the process(es) mentioned.


Is this process the result of an exec?
Yes
Call trace_start() and trace_open_thread() at the beginning because exec'ed processes do not inherit trace mechanisms.
No
Does the parent call trace_start() ?
Yes
Put unique trace_open_thread() call(s) at the beginning of the child process(es) because they inherit the parent's trace mechanism.
No
Put identical trace_start() and unique trace_open_thread() calls at the beginning of each child.

Note: C threads and Ada tasks of the same process may choose to call `trace_open_thread()` and `trace_start()` on their own, however by default the first `trace_start()` and `trace_open_thread()` apply to all C threads and Ada tasks in a given process.

Figure 3-2. `trace_start()` **and** `trace_open_thread()` **Placement**

trace_open_thread()

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

SYNTAX

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

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

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

PARAMETERS

thread_name

In NightTrace every thread of execution to be traced (whether a separate process, or a C thread or Ada task within a process) must be associated with a name, *thread_name*, which may be null. NightTrace's graphical displays and textual summary information show which threads logged trace events. If the `trace_open_thread()` thread name is null, the **`ntrace`** display utility uses the global thread identifier (TID) as a label in these displays. For more information on global thread identifiers see "TID List" on page 8-8.

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

The following words are reserved in NightTrace and should not be used in upper case or lower case as thread names: `NONE, ALL, ALLUSER, ALLKERNEL, TRUE, FALSE, CALC`. See "Pre-Defined String Tables" on page 5-15 for more information about thread names.

NOTE

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

DESCRIPTION

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

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

A process that results from the **`exec(2)`** system service does not inherit a trace mechanism. Therefore, if that process is to log trace events, it must call both `trace_start()` and `trace_open_thread()`. For detailed guidelines on `trace_open_thread()` placement, see Figure 3-2.

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. Solution: Put a `trace_start()` call before the `trace_open_thread()` call.
`[NTINVALID]`	An invalid thread name was specified. Solution: Choose a thread name that meets the requirements mentioned earlier.
`[NTRESOURCE]`	There are not enough resources to open this thread. Solution: Ask your system administrator to increase the size of the process table.
`[NTPGLOCK]`	Permission to lock the text and data pages of the NightTrace library routines was denied. Solution: If `P_PLOCK` privilege is lacking, see your system administrator or invoke **`ntraceud`** with the **`-lockdisable`** option. Note: This can also happen when a forked process changes its user ID to one that does not have page lock privilege, yet its parent process did have page lock privilege.

SEE ALSO

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

See **`intro(2)`** for more information on page lock privilege (`P_PLOCK`). See “ntraceud Options” on page 4-4 for more information on **`ntraceud`** options.

trace_event() and Its Variants

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

SYNTAX

C:

```
int trace_event (int ID);

int trace_event_arg (int ID, long arg);

int trace_event_flt (int ID, float arg);

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

int trace_event_dbl (int ID, double arg);

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

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

Fortran:

```
integer function trace_event (ID)
integer ID

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

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

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

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

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

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

Ada:

```
type event_type is range 0.4095;
```

(procedures)

```
procedure trace_event (ID : event_type);

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

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

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

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

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

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

(functions)

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

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

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

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

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

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

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

PARAMETERS

ID	Each trace event has a user-defined trace event ID, *ID*. This ID is a valid integer in the range reserved for user trace events (`0-4095`, inclusive). See "Pre-Defined String Tables" on page 5-15 for more information about trace event IDs.
argN	Sometimes it is useful to log the current value of a variable or expression, *arg*, along with your trace event. The trace event logging routines provide this capability. They differ by how many and what types of numeric arguments they accept. The `trace_event()` routine takes no *args*. The `trace_event_arg()` routine takes a type long *arg*. The `trace_event_flt()` and `trace_event_two_flt` routines take (floating point) type of float args. The `trace_event_dbl()` and `trace_event_two_dbl()` routines take (floating point) type double *args*. The `trace_event_four_arg()` routine takes four type long *args*. If you want the **`ntrace`** display utility to display these trace event arguments in anything but decimal integer format, you can enter the trace event in an event-map file. See "Understanding Event-Map Files" on page 5-10 for more information on event-map files and formats. Alternatively, you could call the `format()` function. See "format()" on page 9-80 for details. Every call to `trace_event_four_arg()` causes NightTrace to log an `NT_CONTINUE` overhead trace event.

DESCRIPTION

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

TIP:

To save time re-editing, recompiling, and relinking your application, consider beginning with a few too many trace points in the source code. You can then use options to the **`ntraceud`** daemon to selectively enable and disable the logging of specific trace events to the trace event file. See "ntraceud Options" on page 4-4 for more information on **`ntraceud`** options. You can also save time by using **`ntrace`** options to restrict which trace events are loaded for analysis. See "ntrace Options" on page 5-3 for details.

Some typical trace points include the following:

- Suspected bug locations

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

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

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

Probably the most common use of trace events is to identify *states*. Two <u>different</u> 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 **`ntraceud`** 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 **`ntraceud`** 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 3-16 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 StateGraphs, EventGraphs, and DataGraphs in the **`ntrace`** display utility. See “StateGraph” on page 7-14, “EventGraph” on page 7-15, and “DataGraph” on page 7-16 for more information about these display objects.

RETURN VALUES

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

`[NTINVALID]`	An invalid trace event ID has been supplied. Solution: Use trace event IDs only in the range 0-4095, inclusive.
`[NTINIT]`	The NightTrace library routines were not initialized. Solution: Be sure a `trace_start()` and `trace_open_thread()` call precede the trace event logging routine call.
`[NTLOSTDATA]`	The trace event was lost because the shared memory buffer was full. Solution: Do one or more of the following. Increase the trace event capacity of the buffer by invoking **`ntraceud`** with the **`-memsize`** option. Decrease the buffer-full cutoff percentage by invoking **`ntraceud`** with the **`-cutoff`** option. Decrease the **`ntraceud`** sleep interval by invoking **`ntraceud`** with the **`-timeout`** option.

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()`

See Chapter 4 for more information on **`ntraceud`** options.

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 3-11 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 **`ntraceud`** logs so many trace events that it is hard to understand the **`ntrace`** display. Occasionally you know that a particular trace event or trace event range is not interesting at certain times but is interesting at others. When either of these conditions exist, it is useful to disable the extraneous trace events. You can disable trace events temporarily, where you disable and later re-enable them. You can also disable 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 **`ntraceud`** daemon to log to the same trace event file.

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

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

TIP:

Consider invoking **`ntraceud`** with the **`-enable`** and the **`-disable`** options instead of calling the `trace_enable()` and `trace_disable()` routines. Using these options saves you from re-editing, recompiling and relinking your application. See “ntraceud Options” on page 4-4 for more information on **`ntraceud`** options.

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 with `trace_event()` calls or by invoking **`ntraceud`** with the **`-enable`** option.

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_start()` and `trace_open_thread()` call precede the call to the disable or enable routine.
`[NTINVALID]`	An invalid trace event ID has been supplied. Solution: Use trace event IDs only in the range `0-4095`, inclusive.

SEE ALSO

Related routines include:

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

See “ntraceud Options” on page 4-4 for more information on **`ntraceud`** options.

trace_flush() and trace_trigger()

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

SYNTAX

C:
```
int trace_flush();

int trace_trigger();
```

Fortran:
```
integer function trace_flush()

integer function trace_trigger()
```

Ada:
(procedures)
```
procedure trace_flush;

procedure trace_trigger;
```

(functions)
```
    function  trace_flush
return ntrace_error;

    function  trace_trigger
return ntrace_error;
```

DESCRIPTION

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

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

`ntraceud` options let you set limits for the first three conditions above. When you invoke **`ntraceud`** with one of these options and it detects the corresponding condition, it automatically flushes the buffer. See "ntraceud Options" on page 4-4 for more information on **`ntraceud`** options.

There is one key way that `trace_flush()` and `trace_trigger()` differ from the flush control the **`ntraceud`** options provide: with `trace_flush()` and `trace_trigger()` you decide when to asynchronously flush the shared memory buffer based on your program flow, and with certain options **`ntraceud`** flushes the shared memory buffer automatically.

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

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

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

When you run **`ntraceud`** 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 **`ntraceud`** copy the remaining trace events from the shared memory buffer to the trace event file. However, do not call `trace_flush()` or `trace_trigger()` too often or you will reduce the effectiveness of this mode. See "Option to Establish Buffer-Wraparound Mode (-bufferwrap)" on page 4-11 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 **`ntraceud`** daemon; if necessary, restart it and rerun the trace.

SEE ALSO

Related routines include:

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

trace_close_thread()

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

SYNTAX

C: `int trace_close_thread();`

Fortran: `integer function trace_close_thread()`

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

DESCRIPTION

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

RETURN VALUES

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

`[NTINIT]`	The NightTrace library routines were not initialized. Solution: Call `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 trace event tracing in your process.

SYNTAX

C: `int trace_end();`

Fortran: `integer function trace_end()`

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

DESCRIPTION

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

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

When all processes in your application end their respective trace runs, use the following command to flush and close the trace event file.

`ntraceud -quit` *trace_file*

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 **`ntraceud`** daemon; if necessary, restart it and rerun the trace.
`[NTNODAEMON]`	There is no **`ntraceud`** daemon with a trace event file name that matches the one on the `trace_start()` 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. If the trace event file is inexplicably truncated and the **`ntraceud`** daemon is not running, reinvoke **`ntraceud`** and rerun your application.

SEE ALSO

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

Disabling Tracing

There are four ways to disable tracing in your application:

- For C applications, put a `#include <ntrace.h>` in your source code. You must either recompile your application with the **`-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. For more information on preprocessor options, see **`cpp(1)`**.

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

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

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

- Call the `trace_disable_all()` routine near the top of the source, recompile, and relink your application with the NightTrace library. (For more information about this routine, see "trace_enable(), trace_disable(), and Their Variants" on page 3-16.) If your application calls any of the enable routines, this method is not entirely effective.

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

- Start up **`ntraceud`** with the **`-disable 0-4095`** or the **`-enable 0`** option. (At **`ntraceud`** start up, **`-enable 0`** disables all trace events except trace event ID 0. For more information about these options, see "Option to Disable Logging (-disable)" on page 4-22 and "Option to Enable Logging (-enable)" on page 4-24.) If you linked with the NightTrace library before, you do not need to relink.

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

- Do not start up **`ntraceud`**.

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

Compiling and Linking

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

C Example

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

This step:

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

For more information on compiling and linking C programs, see the Concurrent *C Reference Manual.*

Fortran Example

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

This step:

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

For more information on compiling and linking **hf77** programs, see the *hf77 Fortran Reference Manual.*

Ada Example

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

Exercise: Instrumenting Code

Putting library calls in your application is called *instrumenting your code*. The following application is in **`/usr/lib/NightTrace/examples/entry_exit.c`**.

```
#include <sys/types.h>
#include <time.h>
#include <stdio.h>

void take_a_nap( sleep_str )
struct timespec sleep_str;
{
   printf( "Sleeping for %.3f
seconds\n”,
      (float) sleep_str.tv_nsec /
1e+09 );
   nanosleep( &sleep_str, NULL );

   /* make the spacing between states
obvious */
   sleep_str.tv_nsec = 30000000;
   nanosleep( &sleep_str, NULL );
}


main()
{
   int i;
   struct timespec sleep_str;

   for( i=0; i<10; ++i )
   {
      sleep_str.tv_nsec = ( rand() %
1000 ) * 1000000;
      take_a_nap( sleep_str );
   }

   exit( 0 );
}
```

Figure 3-3. entry_exit.c Before Instrumentation

Make a copy of this file in your directory, and call it **`entry_exit.c`**. Make the following changes by inserting trace event library calls at appropriate places in the application:

- Start the trace session and log trace events to a file named **`log`**
- Open a thread named `timings`

- Log trace event `NAP_START` (with trace event ID 10) and the (type `long`) number of nanoseconds to sleep (`sleep_str.tv_nsec`) before the first `nanosleep` call in `take_a_nap`
- Log trace event `NAP_END` (with trace event ID 20) after the first nano-sleep call in `take_a_nap`. (`NAP_START` and `NAP_END` form the boundaries of a state.)
- Close the thread
- End the trace session

An example solution follows.

```
/* For brevity, no return values are
checked */

#include <ntrace.h>
#include <sys/types.h>
#include <time.h>
#include <stdio.h>

#define NAP_START 10
#define NAP_END   20

void take_a_nap( sleep_str )
struct timespec sleep_str;
{
   /* NAP_START & NAP_END are the
boundaries of a state */
   trace_event_arg( NAP_START,
sleep_str.tv_nsec );
   printf( "Sleeping for %.3f
seconds\n”,
      (float) sleep_str.tv_nsec / 1e+09
);
   nanosleep( &sleep_str, NULL );
   trace_event( NAP_END );

   /* make the spacing between states
obvious */
   sleep_str.tv_nsec = 30000000;
   nanosleep( &sleep_str, NULL );
}


main()
{
   int i;
   struct timespec sleep_str;

   trace_start( "log” );
   trace_open_thread( "timings” );

   for( i=0; i<10; ++i )
   {
      sleep_str.tv_nsec = ( rand() %
1000 ) * 1000000;
      take_a_nap( sleep_str );
   }

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

Figure 3-4. entry_exit.c After Instrumentation

This exercise continues in “Exercise: Logging Trace Events” on page 4-27.

4
Generating Trace Event Logs with ntraceud

4
Generating Trace Event Logs with ntraceud

Overview

This chapter describes the following topics:

- The **`ntraceud`** daemon
- The default NightTrace environment
- **`ntraceud`** modes
- **`ntraceud`** options
- Invoking **`ntraceud`**
- Starting your application
- Stopping **`ntraceud`**

The information in this chapter is not pertinent to creating KernelTrace trace event files. For information on creating KernelTrace trace event files, see **`ktrace(1)`** and Chapter 11.

The ntraceud Daemon

When you start up **`ntraceud`**, it creates a daemon background process and returns your prompt. The daemon creates a shared memory buffer in global memory. Your application writes trace events into this buffer, and the daemon copies these trace events to a trace event file.

You supply the name of the trace event file on your **`ntraceud`** invocation and in the `trace_start()` library call in your application. If this file does not exist, **`ntraceud`** creates it; otherwise, **`ntraceud`** overwrites it. Unless your **`umask(1)`** setting overrides this default, **`ntraceud`** creates the file with mode 666, read and write permission to all users. If you want to maximize performance, use a trace event file that is local to the system where the **`ntraceud`** daemon and your application run.

A single **`ntraceud`** daemon may service several running applications or processes. Several **`ntraceud`** daemons can run simultaneously; the system identifies them by their distinctive trace event file names. The **`ntraceud`** daemon resides on your system under**`/usr/bin/ntraceud`**.

You must invoke **ntraceud** before any process in your application initializes a trace by calling the `trace_start()` library routine. See "trace_start()" on page 3-5 for more information.

Whenever the daemon is idle, it sleeps. You can control the sleep interval with an **ntraceud** option. Logging a trace event may wake the daemon if the buffer-full cutoff percentage is exceeded or if shared memory becomes full of trace events. Flushing trace events from the shared memory buffer to disk always wakes the daemon.

The Default NightTrace Environment

You enter the default NightTrace environment by invoking **ntraceud** with a trace event file argument and without any options. You can override defaults by invoking **ntraceud** with particular options. Table 4-1 summarizes these options. Later sections provide detailed descriptions of these options and operating modes.

In the default environment, all trace events are enabled for logging. Your application logs trace events to the shared memory buffer. By default, the interval timer (NightHawk 6000 Series) or the Time Base Register (Power Hawk/PowerStack) is used to timestamp trace events. However, the user may change the event timestamp source using the **-clock** option to **ntraceud** (see "Option to Select Timestamp Source (-clock)" on page 4-17).

The **ntraceud** daemon operates in *expansive mode*. In expansive mode, **ntraceud** copies all trace events from the shared memory buffer to the trace event file. This behavior differs from file-wraparound mode and buffer-wraparound mode. If the trace event file does not exist when **ntraceud** starts up, **ntraceud** creates it; otherwise, **ntraceud** overwrites it.

ntraceud and the NightTrace library routines use page locking to prevent page faults during trace event logging. NightTrace also modifies the shared memory region bound to the system's interrupt priority level (IPL) register; this action prevents rescheduling and interrupts during trace event logging.

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

- **ntraceud**'s sleep interval elapses
- The buffer-full cutoff percentage is exceeded
- The shared memory buffer is full of unwritten trace events
- Your application calls `trace_flush()`, `trace_trigger()`, or `trace_end()`

A summary of NightTrace environment defaults follows.

Table 4-1. NightTrace Environmental Defaults

Characteristic	Default	Modifying Option
`ntraceud` sleep interval	5 seconds	**`-timeout`** *seconds*
Buffer-full cutoff percentage	20% full	**`-cutoff`** *percent*
Shared memory buffer size	16K (16,384) trace events	**`-memsize`** *count*
Flush mechanism	(See above)	**`-bufferwrap`**
Trace event file size	Indefinite	**`-filewrap`** *bytes*
Trace events enabled for logging	All	**`-disable`** *ID* and **`-enable`** *ID*
Page-fault handling	Page locking	**`-lockdisable`**
Interrupt handling	Modify IPL register	**`-ipldisable`**

ntraceud Modes

NightTrace can operate in three modes: expansive (default), file-wraparound, and buffer-wraparound. As the following two tables show, these modes meet different needs and have different characteristics. They differ mainly by their handling of the shared memory buffer and the trace event file on disk.

By default, NightTrace operates in expansive mode. NightTrace operates in file-wraparound mode when you specify the **`-filewrap`** option on the **`ntraceud`** invocation line. The **`ntraceud -bufferwrap`** option puts NightTrace in buffer-wraparound mode. See “Option to Establish File-Wraparound Mode (-filewrap)” on page 4-10 and “Option to Establish Buffer-Wraparound Mode (-bufferwrap)” on page 4-11 for more information on these options.

It is not possible to combine expansive mode with either file-wraparound or buffer-wraparound mode. Although you can mix file-wraparound and buffer-wraparound modes, it is not recommended.

Table 4-2 provides some guidelines to help you decide which mode to use.

Table 4-2. Mode-Selection Guidelines

	MODE		
Constraint	Expansive	File-Wraparound	Buffer-Wraparound
Trace event importance	All trace events are important	Newest trace events are important	Events just before a `trace_flush()` are important
General	Disk space and memory are plentiful	Disk space is limited	Program will run a long time

Table 4-3 shows how each NightTrace operating mode reacts to a particular condition. The process of copying trace events from the shared memory buffer to the trace event file on disk is called *flushing the buffer.*

Table 4-3. NightTrace Operating Modes

	MODE		
Condition	Expansive	File-Wraparound	Buffer-Wraparound
`ntraceud` sleep interval exceeded (**`-timeout`**)	Flush the buffer	Flush the buffer	(No reaction)
Buffer-full cutoff percentage exceeded (**`-cutoff`**)	Flush the buffer	Flush the buffer	(No reaction)
Shared memory buffer is full (**`-memsize`**)	Flush the buffer	Flush the buffer	Overwrite the buffer's oldest trace events with the newest ones
Trace event file is full (**`-filewrap`**)	N/A	Overwrite the file's oldest trace events with the newest ones	N/A

ntraceud Options

`ntraceud` always copies trace events from the shared memory buffer to the trace event file, *trace_file*. You can override some other NightTrace defaults by invoking **`ntraceud`**

with option(s). You can also use options to quit running or reset **`ntraceud`** and to obtain version, statistical, or invocation-syntax information. The full **`ntraceud`** invocation syntax is:

> **`ntraceud`** [**`-help`**] [**`-version`**] [**`-ipldisable`**] [**`-lockdisable`**]
> [**`-filewrap`** *bytes*] [**`-bufferwrap`**] [**`-memsize`** *count*]
> [**`-timeout`** *seconds*] [**`-cutoff`** *percent*] [**`-clock`** *source*]
> [**`-reset`**] [**`-quit`**] [**`-stats`**] [[**`-disable`** *ID*[**`-`***ID*]] [...]]
> [[**`-enable`** *ID*[**`-`***ID*]] [...]] *trace_file*

You can abbreviate all **`ntraceud`** options to their shortest unambiguous length, but most of the examples in this manual use the long option name. These options are case-insensitive. The following examples are all equivalent:

```
ntraceud -help
ntraceud -hel
ntraceud -he
ntraceud -h
ntraceud -H
ntraceud -HE
ntraceud -Hel
ntraceud -HELP
```

You can invoke **`ntraceud`** more than once with different options during a single trace session; each invocation passes additional options and values to the running **`ntraceud`** daemon. Usually you do this to dynamically enable or disable trace events or to obtain current statistical information. Options that are available only at **`ntraceud`** start up are described that way.

The following sections discuss the **`ntraceud`** options.

Option to Get Help (-help)

The **`ntraceud -help`** option displays the **`ntraceud`** invocation syntax on standard output.

SYNTAX

`ntraceud -help`

DESCRIPTION

The **`ntraceud -help`** option displays a brief help message showing the complete invocation syntax for **`ntraceud`**. Screen 4-1 shows an example of **`-help`** option output.

```
usage:  ntraceud  [-help]  [-version]  [-ipldisable]  [-lockdisable]
   [-filewrap bytes]  [-bufferwrap]  [-memsize count]  [-timeout seconds]
   [-cutoff percent]  [-clock source]  [-reset]  [-quit]  [-stats]
   [-disable ID[-ID]]  [-enable ID[-ID]]  trace_file

General options:
   -help             Write this message to standard output
   -version          Write the current ntraceud version stamp to standard
output

Options for a new ntraceud daemon:
   -ipldisable       Disable use of the IPL register
   -lockdisable      Disable use of page locking
   -filewrap bytes   Use file wraparound mode with max trace_file size in bytes
   -bufferwrap       Use shared memory buffer wraparound mode
   -memsize count    Set shared memory buffer size to specified event count
   -timeout seconds  Set the ntraceud timeout to specified seconds
   -cutoff percent   Flush events to disk at specified cutoff level
   -clock source     Specify source of event time stamps
      Valid values for source are:
         default        Use the default system clock
         rcim_tick      Use the RCIM synchronized tick clock

Options for an existing ntraceud daemon:
   -reset            Reset the ntraceud daemon and the trace_file
   -quit             Quit running ntraceud
   -stats            Write statistics (resource/environment) to standard output

Options for new and existing ntraceud daemons:
   -disable ID[-ID]  Disable a specific event ID or ID range from logging
   -enable  ID[-ID]  Enable a specific event ID or ID range to log

Files:
   trace_file        Holds events logged by your application and ntraceud
```

Screen 4-1. Sample Output from the ntraceud -help Option

Option to Get Version Information (-version)

The **`ntraceud -version`** option displays the current **`ntraceud`** version stamp on standard output.

SYNTAX

```
ntraceud -version
```

DESCRIPTION

The **`ntraceud -version`** option displays version stamp information for this **`ntraceud`** daemon.

Option to Disable the IPL Register (-ipldisable)

The **`ntraceud -ipldisable`** option disables the default use of the system's interrupt priority level (IPL) register by **`ntraceud`** and by the NightTrace library routines in your application.

SYNTAX

`ntraceud -ipldisable` *trace_file*

DESCRIPTION

You can identify a running **`ntraceud`** daemon by its trace event file name, *trace_file*.

By default, NightTrace modifies a shared memory region bound to the system's interrupt priority level (IPL) register. This modification prevents rescheduling and interrupts during trace event logging.

If your application lacks read and write privilege to **`/dev/spl`**, the NightTrace daemon and library initialization routine exit with errors. If you still want to trace events, you must invoke the **`ntraceud`** daemon with the **`-ipldisable`** option. Note, however, that rescheduling and interrupts may distort trace event timings. If you use the **`-ipldisable`** option, you must start up **`ntraceud`** with it.

You must not use the **`-ipldisable`** option if your user-level interrupt routine logs trace events to the shared memory buffer.

CAUTION

The **`-ipldisable`** option should be used with great care to avoid deadlock. This may occur if more than one LWP, each biased to run on the same CPU, is logging trace events to a trace file created by an **`ntraceud`** invoked with the **`-ipldisable`** option.

Consider the following scenario: an LWP, preparing to log a trace event, locks the spin lock to protect the shared memory buffer. It is preempted by a second LWP which also attempts to log a trace event. However, due to priority inversion, the first LWP cannot release the spin lock, causing the second LWP to loop infinitely waiting for the spin lock to be released.

This deadlock could be avoided if **`ntraceud`** were invoked without the **`-ipldisable`** option. This would allow the first LWP to release the spin lock before being preempted.

SEE ALSO

For more information on the IPL register, see the *PowerMAX OS Programming Guide*.

Option to Prevent Page Locking (-lockdisable)

The **`ntraceud -lockdisable`** option disables default page locking by **`ntraceud`** and by the NightTrace library routines in your application.

SYNTAX

`ntraceud -lockdisable` *trace_file*

DESCRIPTION

You can identify a running **`ntraceud`** daemon by its trace event file name, *trace_file*.

By default, NightTrace locks its pages in memory. This capability prevents page faults during trace event logging that could distort trace event timings.

If you lack `P_PLOCK` privilege needed to lock your pages in memory, your invocation of **`ntraceud`** and your application exit with errors. If you still want to trace events, you must invoke the **`ntraceud`** daemon with the **`-lockdisable`** option. This option makes **`ntraceud`** and the NightTrace library routines in your application run without locking their pages in memory. Note, however, that page faults may distort trace event timings. If you use the **`-lockdisable`** option, you must start up **`ntraceud`** with it.

You must <u>not</u> use the **`-lockdisable`** option if your user-level interrupt routine logs trace events to the shared memory buffer.

SEE ALSO

For more information on privileges, see “Administering Privileges” on page 2-4.

Option to Establish File-Wraparound Mode (-filewrap)

By default, the trace event file can grow indefinitely. With the **`ntraceud -filewrap`** option, you can make NightTrace operate in file-wraparound mode, rather than expansive mode. In file-wraparound mode, you limit the trace event file size.

SYNTAX

`ntraceud -filewrap` *bytes trace_file*

DESCRIPTION

The **`ntraceud -filewrap`** option lets you specify the maximum byte size, *bytes*, of the trace event file, *trace_file*. Specify the *bytes* parameter as a number of bytes or as a number with a K or M suffix to show that the *bytes* parameter is in kilobyte or megabyte units, respectively. For example, 12K means 12,288 bytes. If you use the **`-filewrap`** option, you must start up **`ntraceud`** with it.

Your application logs enabled trace events into a shared memory buffer. **`ntraceud`** copies these trace events to the trace event file. In expansive mode, this file can grow indefinitely.

The **`ntraceud -filewrap`** option makes NightTrace operate in file-wraparound mode, rather than in expansive mode. In file-wraparound mode the trace event file can become full of trace events. When this happens, **`ntraceud`** overwrites the oldest trace events in the beginning of the file with the newest ones, intentionally discarding the oldest trace events to make room for the newest ones.

In expansive (default) and file-wraparound modes, you control automatic buffer flushing by setting the **`ntraceud`** sleep interval, shared memory size, and buffer-full cutoff percentage. In contrast, there is no automatic buffer flushing in buffer-wraparound mode; these values have no effect in this mode.

File-wraparound mode can be beneficial if you are short of disk space. With this mode, you specify the maximum size of the trace event file, instead of allowing it to grow indefinitely. Consider using this option if you are interested only in the most recent of many trace events logged by an application over a long period of time. If you want to determine how much disk space is available, run the **`df(1)`** command with the **`-k`** option and look at the "`avail`" column.

SEE ALSO

For a comparison of expansive, file-wraparound, and buffer-wraparound modes, see "ntraceud Modes" on page 4-3.

Option to Establish Buffer-Wraparound Mode (-bufferwrap)

The process of copying trace events from the shared memory buffer to the trace event file on disk is called *flushing the buffer*. With the **`ntraceud -bufferwrap`** option, you can make NightTrace operate in buffer-wraparound mode, rather than expansive mode. In buffer-wraparound mode, the **`ntraceud`** daemon flushes only the most recent trace events, rather than all trace events. Your application asynchronously triggers every buffer flush.

SYNTAX

`ntraceud -bufferwrap` *trace_file*

DESCRIPTION

The **`ntraceud`** daemon always logs enabled trace events into a shared memory buffer. In expansive mode, when the buffer is full (or when some other conditions exist), **`ntraceud`** automatically flushes the buffer to the trace event file, *trace_file*.

The **`ntraceud -bufferwrap`** option makes NightTrace operate in buffer-wraparound mode, rather than in expansive mode. When the buffer is full in buffer-wraparound mode, the application treats the shared memory buffer as a circular queue and overwrites the oldest trace events with the newest ones, intentionally discarding the oldest trace events to make room for the newest ones. This overwriting continues until your application explicitly calls `trace_flush()`. Only then, does **`ntraceud`** copy the remaining trace events from the shared memory buffer to the trace event file. If you use the **`-bufferwrap`** option, you must start up **`ntraceud`** with it.

NOTE

You control automatic buffer flushing by setting the **`ntraceud`** sleep interval and buffer-full cutoff percentage in expansive (default) mode and in file-wraparound mode. In contrast, there is no automatic buffer flushing in buffer-wraparound mode; these values have no effect in this mode. Invoking **`ntraceud`** with the **`-bufferwrap`** option, makes **`ntraceud`** ignore any **`-timeout`** and **`-cutoff`** options.

In buffer-wraparound mode, you can estimate the maximum number of trace events to be written to your trace event file by using the following formula:

```
max_events = max_events_in_buffer * flush_count
```

where:

`max_events`	The maximum number of trace events.

`max_events_in_buffer`
: The number of trace events the shared memory buffer can hold. You can set this value when you invoke **`ntraceud`** with the **`-memsize`** option.

`flush_count`
: The number of `trace_flush()` calls your application executes.

For example, if you set your shared memory buffer size to 1000 trace events, then `max_events_in_buffer` is 1000. If you expect your three `trace_flush()` calls to execute two times each, then `flush_count` is six (3 * 2). Calculating `max_events` gives you about 6000 (1000 * 6) trace events in your trace event file.

Buffer-wraparound mode:

- Can help you with debugging
- Can reduce trace events to a manageable number
- May conserve disk space

Buffer-wraparound mode can be useful in debugging.

Assume that you are debugging a fault in a large application. Follow the steps below to accomplish your task.

1. Insert a `trace_flush()` call in your code where you believe the fault occurs.
2. Compile and link your application.
3. Invoke **`ntraceud`** with the **`-bufferwrap`** option.
4. Run your application.

When your application executes the `trace_flush()` call, **`ntraceud`** copies all trace events still in the shared memory buffer to the trace event file. You can then use the **`ntrace`** display utility to graphically analyze only the trace events immediately preceding the fault.

Buffer-wraparound mode can also be useful in reducing trace events to a manageable number. In this mode, when the shared memory buffer is full, the newest trace events overwrite the oldest ones. This means that if the shared memory buffer becomes full before your application executes the `trace_flush()` call, **`ntraceud`** copies only the current contents of the buffer to the trace event file. This way, you can exclude the oldest trace events from your **`ntrace`** displays.

In buffer-wraparound mode, **`ntraceud`** usually flushes fewer trace events to the trace event file than in expansive mode. Thus, this mode can conserve disk space.

If you want to determine how much disk space is available, run the **`df(1)`** command with the **`-k`** option and look at the "`avail`" column. Use the following command to see the system settings for the current, default, minimum, and maximum shared memory segment size:

```
$ /etc/conf/bin/idtune -g SHMMAX
```

See the **`idtune(1M)`** man page for more information.

SEE ALSO

For more information on `trace_flush()`, see “trace_flush() and trace_trigger()” on page 3-20. For a comparison of expansive, file-wraparound, and buffer-wraparound modes, see “ntraceud Modes” on page 4-3. For information on limiting the number of logged trace events, see “Option to Define Shared Memory Buffer Size (-memsize)” on page 4-14.

Option to Define Shared Memory Buffer Size (-memsize)

By default, the shared memory buffer can hold 16,384 trace events. When the buffer is full of unwritten trace events, the **`ntraceud`** daemon wakes up and copies the trace events to the trace event file. The **`ntraceud -memsize`** option lets you alter the size of the shared memory buffer.

SYNTAX

`ntraceud -memsize` *count trace_file*

DESCRIPTION

The **`ntraceud -memsize`** option lets you set the shared memory buffer size. Specify the *count* parameter as a maximum number of trace events or as a number with a K or M suffix to show that the *count* parameter is in kilobyte or megabyte units, respectively. For example, 12K means 12,288 trace events. **`ntraceud`** rounds that number up to a full page boundary. A trace event with zero or one argument takes up 16 bytes; a trace event with more arguments takes up 32 bytes: 16 bytes for the basic trace event and one argument and 16 bytes for the `NT_CONTINUE` overhead trace event and the remaining arguments.

Use the following command to see the system settings for the current, default, minimum, and maximum shared memory segment size:

```
$ /etc/conf/bin/idtune -g SHMMAX
```

See the **`idtune(1M)`** man page for more information.

By default, if the shared memory buffer becomes full, **`ntraceud`** wakes up and copies trace events from the shared memory buffer to the trace event file, *trace_file*. You can increase the *count* parameter to prevent trace event loss. If you use the **`-memsize`** option, you must start up **`ntraceud`** with it.

By changing the shared memory buffer size, you can:

- Alter the buffer flush frequency
- Control the number of trace events copied to the trace event file in buffer-wraparound mode

SEE ALSO

For information on limiting the number of logged trace events, see “Option to Establish Buffer-Wraparound Mode (-bufferwrap)” on page 4-11.

Option to Set Timeout Interval (-timeout)

By default, **`ntraceud`** sleeps 5 seconds after writing trace events to disk. The **`ntraceud -timeout`** option lets you set this timeout interval.

SYNTAX

`ntraceud -timeout` *seconds trace_file*

DESCRIPTION

You can identify a running **`ntraceud`** daemon by its trace event file name, *trace_file*.

When **`ntraceud`** is idle, the daemon sleeps. By default, the sleep interval is a maximum of 5 seconds. The **`ntraceud -timeout`** option lets you establish the maximum number of seconds, *seconds*, that the **`ntraceud`** daemon sleeps.

Waking the **`ntraceud`** daemon incurs overhead that can distort trace event timings; decreasing the timeout parameter makes it more likely that the daemon will be awake when needed. You can also decrease the timeout parameter to prevent trace event loss. Note: If your application does not log events frequently, you can increase the timeout to reduce the time the daemon runs and consumes CPU cycles.

If you use the **`-timeout`** option, you must start up **`ntraceud`** with it. If you invoke **`ntraceud`** with both the **`-timeout`** and **`-bufferwrap`** options, **`ntraceud`** ignores the **`-timeout`** option.

`ntraceud` does not sleep for the full period if:

- Your application executes a call to `trace_flush()`, `trace_trigger()`, or `trace_end()`
- Your application logs a trace event that causes shared memory to become full or your buffer-full cutoff percentage to be reached
- You specify a timeout parameter which exceeds the time in which the lower 32 bits of the timestamp source would roll over. This rollover time varies from architecture to architecture (with a minimum value of 257.69803 seconds) and is calculated by **`ntraceud`** as part of its initialization. It is important to detect this rollover so that proper ordering of trace events is maintained. If you specify a timeout interval which exceeds the rollover time, **`ntraceud`** uses the rollover time as the timeout interval, ignoring the value specified.

Option to Set the Buffer-Full Cutoff Percentage (-cutoff)

By default, when the shared memory buffer becomes 20-percent full of unwritten trace events, the **`ntraceud`** daemon wakes up and copies the trace events to the trace event file. The **`ntraceud -cutoff`** option lets you alter this percentage.

SYNTAX

`ntraceud -cutoff` *percent trace_file*

DESCRIPTION

The **`ntraceud -cutoff`** option lets you set the buffer-full cutoff percentage, *percent*, for the shared memory buffer. *percent* is an integer percentage in the range 0-99, inclusive.

The process of copying trace events from the shared memory buffer to the trace event file, *trace_file*, on disk is called *flushing the buffer*. When a logged trace event causes the buffer to reach the buffer-full cutoff percentage, **`ntraceud`** wakes up and flushes the buffer.

Waking the **`ntraceud`** daemon incurs overhead that can distort trace event timings; decreasing the shared memory buffer-full cutoff percentage makes it more likely that the daemon will be wakened by the application. You can also decrease the *percent* parameter to prevent trace event loss; the effect is an increase in the buffer flush frequency.

If you use the **`-cutoff`** option, you must start up **`ntraceud`** with it. If you invoke **`ntraceud`** with both the **`-cutoff`** and **`-bufferwrap`** options, **`ntraceud`** ignores the **`-cutoff`** option.

Option to Select Timestamp Source (-clock)

The **`ntraceud -clock`** option allows you to select which timing source will be used to timestamp events.

SYNTAX

`ntraceud -clock` *source trace_file*

DESCRIPTION

The **`ntraceud -clock`** option lets you select the timing source used to timestamp trace events. Valid *source* values are:

`default`	the interval timer (NightHawk 6000 Series) or the Time Base Register (Power Hawk/PowerStack)
`rcim_tick`	the RCIM synchronized tick clock

If you invoke **`ntraceud`** with the **`-clock`** option, you must supply a value for the *source*.

If **`rcim_tick`** is specified for the *source* and the system on which you are tracing does not have an RCIM installed or configured or if the RCIM synchronized tick clock on the system on which you are tracing is stopped, the NightTrace daemon and library initialization routine exit with errors.

If the **`-clock`** option is not specified, the interval timer (NightHawk 6000 Series) or the Time Base Register (Power Hawk/PowerStack) is used to timestamp trace events.

Option to Reset the ntraceud Daemon (-reset)

The **`ntraceud -reset`** option resets a running **`ntraceud`** daemon process.

SYNTAX

`ntraceud -reset` *trace_file*

DESCRIPTION

You can identify a running **`ntraceud`** daemon by its trace event file name, *trace_file*. By default, **`ntraceud`** overwrites the trace event file if it is not currently in use. In contrast, the **`ntraceud -reset`** option empties the file and prepares the running daemon for another trace run. Use the **`-reset`** option when you are no longer interested in the contents of an active trace event file. You can invoke **`ntraceud`** multiple times with the **`-reset`** option.

SEE ALSO

For information on quitting an `ntraceud` session without clearing the trace event file, see “Option to Quit Running ntraceud (-quit)” on page 4-19.

Option to Quit Running ntraceud (-quit)

The **`ntraceud -quit`** option terminates a running **`ntraceud`** process.

SYNTAX

`ntraceud -quit` *trace_file*

DESCRIPTION

You can identify a running **`ntraceud`** daemon by its trace event file name, *trace_file*.

A process completes its NightTrace session by calling `trace_end()` or exiting normally. The **`-quit`** option tests whether all processes dealing with a particular running **`ntraceud`** daemon have completed trace event logging. If some processes have not completed, **`ntraceud`** waits. If all processes have completed, the option:

- Terminates the running daemon process
- Flushes remaining trace events to the trace event file
- Closes the file
- Removes the shared memory buffer

TIP:

You cannot get statistical information after you quit running **`ntraceud`**. Consider getting statistical information before you quit running **`ntraceud`**. For statistical information on your trace session, see “Option to Present Statistical Information (-stats)” on page 4-20.

Assume that you have invoked **`ntraceud`** with the **`-quit`** option, and you want to reinvoke **`ntraceud`** with the same trace event file. Your next **`ntraceud`** invocation will automatically overwrite the trace event file.

SEE ALSO

For information on resetting **`ntraceud`** and the trace event file for another session, see “Option to Reset the ntraceud Daemon (-reset)” on page 4-18.

Option to Present Statistical Information (-stats)

The **`ntraceud -stats`** option presents a display of statistical information for a running **`ntraceud`** daemon on standard output.

SYNTAX

`ntraceud -stats` *trace_file*

DESCRIPTION

You can identify a running **`ntraceud`** daemon by its trace event file name, *trace_file*.

The **`-stats`** option provides statistical information that tells you about your current NightTrace environment and resource use. This information can help you determine if you have adequate resources for your application. If you are interested in watching changes in the statistics, invoke **`ntraceud`** multiple times with the **`-stats`** option.

Specifically, the **`-stats`** option provides information on:

- **`ntraceud`** mode. **`ntraceud`** may run in the following modes:
  - `NT_M_DEFAULT`, meaning expansive (default) mode
  - `NT_M_FILEWRAP`, meaning file-wraparound mode
  - `NT_M_BUFFERWRAP`, meaning buffer-wraparound mode
- Shared memory buffer size
- Buffer-full cutoff percentage
- **`ntraceud`** timeout interval
- Number of threads or processes logging in your application
- Number of times trace events were lost. This statistic refers to a situation that infrequently arises during a NightTrace session. **`ntraceud`** may lose some trace events if the trace events enter the shared memory buffer faster than **`ntraceud`** can copy them to the trace event file. For more information on this topic, see “Preventing Trace Events Loss” on page A-1.
- Number of automatic buffer flushes (For more information on buffer flushes, see “trace_flush() and trace_trigger()” on page 3-20.)
- Number of trace events logged to shared memory. **`ntraceud`** and some NightTrace library routines occasionally log predefined trace events into the shared memory buffer. Therefore, the statistic for number of trace events logged to shared memory may exceed the number of times your application logs a trace event.
- Trace event IDs enabled

Screen 4-2 shows a sample of **`-stats`** option output.

```
$ ntraceud -stats log

NTRACEUD STATISTICS

The ntraceud daemon is running in NT_M_DEFAULT mode.
There is a maximum of 16384 trace events in the shared memory buffer
The buffer-full threshold is 20% or 3276 trace events
The daemon timeout period is 5 seconds
There are 1 thread(s) logging trace events
The shared memory buffer had 0 events lost
There have been 0 unrequested buffer flushes
The total number of trace events logged to shared memory is 5

Enabled Events:
0-4095
```

Screen 4-2. Sample Output from ntraceud -stats Option

Defaults for some of these values exist in the header file **`/usr/include/ntrace.h`**. You can override the default values with **`ntraceud`** options. See Table 4-1 for more information on the default values and the corresponding options used to override them.

SEE ALSO

For information on trace event loss prevention, see “Option to Establish File-Wrap-around Mode (-filewrap)” on page 4-10, “Option to Set Timeout Interval (-timeout)” on page 4-15, and “Option to Set the Buffer-Full Cutoff Percentage (-cutoff)” on page 4-16.

Option to Disable Logging (-disable)

By default, all trace events are enabled for logging to the shared memory buffer. The **`ntraceud -disable`** option makes the application ignore requests to log a specific trace event or range of trace events.

SYNTAX

> **`ntraceud -disable`** *ID* [...] *trace_file*
> **`ntraceud -disable`** *ID_low*-*ID_high* [...] *trace_file*

DESCRIPTION

Sometimes **`ntraceud`** logs so many trace events that it is hard to understand the **`ntrace`** display. Occasionally you know that a particular trace event or trace event range is not interesting at certain times but is interesting at others. When either of these conditions exist, it is useful to disable the extraneous trace events. You can disable trace events temporarily, where you disable and later re-enable them. You can also disable trace events permanently, where you disable them before the application runs or during its execution and never re-enable them.

In the first format, the **`ntraceud -disable`** option dynamically disables a specific trace event ID, *ID*, from logging to the shared memory buffer. In the second format, the **`ntraceud -disable`** option dynamically disables a range of trace event IDs, *ID_low* through *ID_high*, from logging to the shared memory buffer. In either case, trace event IDs are integers in the range 0-4095, inclusive. At defined times, **`ntraceud`** copies trace events from the shared memory buffer to the trace event file, *trace_file*.

NOTE

> The **`-disable`** option disables trace events in all processes that rely on the same **`ntraceud`** daemon to log to the same trace event file.

This first format provides the same functionality as the `trace_disable()` NightTrace library routine. The second format provides the same functionality as the `trace_disable_range()` NightTrace library routine. One advantage of using the **`-disable`** option rather than the library routine is that you do not have to re-edit, recompile, and relink your application. For more information on disable library routines, see “trace_enable(), trace_disable(), and Their Variants” on page 3-16.

Note: In the following text, the names of the trace event files are varied for interest.

You can start up **`ntraceud`** with the **`-disable`** (**`-d`**) option. You can also re-invoke **`ntraceud`** with this option while **`ntraceud`** is running. Furthermore, using the **`-disable`** option to disable an already disabled trace event has no effect. For example, assume that you invoke **`ntraceud`** three times, sequentially, before your application terminates and that **`ntraceud`** has not logged to the `ntoutput` file before.

$ **`ntraceud -d4 ntoutput`** -- *trace event 4 is disabled*
$ **`ntraceud -d7 ntoutput`** -- *trace events 4 & 7 are now disabled*
$ **`ntraceud -d4 ntoutput`** -- *no effect; trace events 4 & 7 disabled*

There may be any number of **`-disable`** options on an **`ntraceud`** invocation line. The following example illustrates this fact.

$ **`ntraceud -d10 -d15 mytrace`** -- *trace events 10 & 15 are disabled*

You may specify a hyphenated trace event range on the **`ntraceud`** invocation line. The following example depicts this case.

$ **`ntraceud -d23-25 traceoutput`** -- *events 23, 24, and 25 disabled*

The following two sequences show how important timing can be when you use the **`-disable`** option. The same steps appear in both sequences, but their order differs. When the first sequence ends, nothing has been logged and all trace events are enabled. In contrast, when the second sequence ends, trace event 52 has been logged once and is now disabled.

Table 4-4. ntraceud Disable Sequence #1

From the Shell	From the Application	Comments
1. Invoke **`ntraceud`**		All trace events enabled
2. Invoke **`ntraceud -d52`**		Trace event 52 disabled
3. Start application		
4.	Call `trace_event(52)`	Trace event 52 <u>not</u> logged
5.	Call `trace_enable(52)`	Trace event 52 enabled

Table 4-5. ntraceud Disable Sequence #2

From the Shell	From the Application	Comments
1. Invoke **`ntraceud`**		All trace events enabled
2. Start application		Trace event 52 enabled
3.	Call `trace_event(52)`	Trace event 52 logged
4.	Call `trace_enable(52)`	No effect
5. Invoke **`ntraceud -d52`**		Trace event 52 disabled

SEE ALSO

For information on enabling trace events, see “Option to Enable Logging (-enable)” on page 4-24 and “trace_enable(), trace_disable(), and Their Variants” on page 3-16.

Option to Enable Logging (-enable)

By default, all trace events are enabled for logging to the shared memory buffer. The **`ntraceud -enable`** option makes the application notice previously disabled requests to log a specific trace event or a range of trace events.

SYNTAX

> **`ntraceud -enable`** *ID* [...] *trace_file*
> **`ntraceud -enable`** *ID_low*-*ID_high* [...] *trace_file*

DESCRIPTION

In the first format, the **`ntraceud -enable`** option dynamically re-enables a specific disabled trace event ID, *ID*, for logging to the shared memory buffer. In the second format, the **`ntraceud -enable`** option dynamically re-enables a range of disabled trace event IDs, *ID_low* through *ID_high*, for logging to the shared memory buffer. In either case, trace event IDs are integers in the range 0-4095, inclusive. At defined times, **`ntraceud`** copies trace events from the shared memory buffer to the trace event file, *trace_file*.

NOTE

The **`-enable`** option affects all processes that rely on the same **`ntraceud`** daemon to log to the same trace event file.

The first format provides the same functionality as the `trace_enable()` NightTrace library routine. The second format provides the same functionality as the `trace_enable_range()` NightTrace library routine. One advantage of using the **`ntraceud`** option instead of the library routine is that you do not have to re-edit, recompile, and relink your application. For more information on enable library routines, see “trace_enable(), trace_disable(), and Their Variants” on page 3-16.

In the following text, the names of the trace event files are varied for interest. Unless otherwise stated, all the following examples describe the results of a non-startup **`ntraceud`** invocation.

There may be any number of **`-enable`** (**`-e`**) options on an **`ntraceud`** invocation line. The following example illustrates this fact.

```
$ ntraceud -e10 -e15 mytrace -- trace events 10 and 15 enabled
```

You may specify a hyphenated trace event range on the **`ntraceud`** invocation line. The following example depicts this case.

```
$ ntraceud -e23-25 traceoutput -- trace events 23, 24, & 25
                                          enabled
```

The **-enable** option acts differently when you use it:

- On **ntraceud** start up
- On later **ntraceud** invocations

If you start up **ntraceud** with the **-enable** option, the specified trace event(s) are the only one(s) enabled; all other trace events are disabled. For example, if the following invocation starts up **ntraceud**, then only trace event 18 is enabled.

```
$ ntraceud -e18 traceout
```

When you use the **-enable** option on non-startup **ntraceud** invocations, NightTrace adds the specified trace event(s) to the list of enabled trace events. Furthermore, attempting to enable an already enabled trace event has no effect. For example, assume that you invoke **ntraceud** four times, sequentially, before your application terminates and that **ntraceud** has not logged to the `ntoutput` file before.

```
$ ntraceud ntoutput          -- all trace events enabled
$ ntraceud -d4 -d7 ntoutput  -- all except 4 and 7 enabled
$ ntraceud -e4 ntoutput      -- all except 7 enabled
$ ntraceud -e4 ntoutput      -- no effect; all except 7 enabled
```

The following two sequences show how important timing can be when you use the **-enable** option. The same steps appear in both sequences, but their order differs. When the first sequence ends, nothing has been logged and all trace events are enabled. In contrast, when the second sequence ends, trace event 52 has been logged once and is now disabled.

Table 4-6. ntraceud Enable Sequence #1

	From the Shell	From the Application	Comments
1.	Invoke **ntraceud**		All trace events enabled
2.	Start application		
3.		Call `trace_disable(52)`	Trace event 52 disabled
4.		Call `trace_event(52)`	Trace event 52 <u>not</u> logged
5.	Invoke **ntraceud -e52**		Trace event 52 enabled

Table 4-7. ntraceud Enable Sequence #2

	From the Shell	From the Application	Comments
1.	Invoke **ntraceud**		All trace events enabled
2.	Start application		
3.		Call `trace_event(52)`	Trace event 52 logged
4.	Invoke **ntraceud -e52**		No effect
5.		Call `trace_disable(52)`	Trace event 52 disabled

SEE ALSO

For information on disabling trace events, see "Option to Disable Logging (-disable)" on page 4-22 and "trace_enable(), trace_disable(), and Their Variants" on page 3-16.

Invoking ntraceud

Now that your system and user environment support NightTrace and you understand the **`ntraceud`** options, you can start up **`ntraceud`**. This section shows a few common **`ntraceud`** invocation examples. In each example, the *trace_file* argument corresponds to the trace event file name you supply on your call to the `trace_start()` library routine.

Normally, your first **`ntraceud`** invocation looks something like the following sample.

`ntraceud` *trace_file*

The next sample invocation assumes that you lack both page lock privilege (**`-lockdisable`**) and read and write access to **`/dev/spl`** needed to modify the interrupt priority level register (**`-ipldisable`**).

`ntraceud -lockdisable -ipldisable` *trace_file*

You may use an invocation similar to the following one if you are tuning your NightTrace environment because you lost trace events last time.

`ntraceud -memsize` *count* **`-timeout`** *seconds* **`-cutoff`** *percent trace_file*

There are several times when you may want to use the following invocation. Usually this invocation is appropriate if you are using `trace_flush()` calls to debug a fault in your application or to reduce the number of logged trace events so the **`ntrace`** display is more readable.

`ntraceud -bufferwrap` *trace_file*

The following invocation is also useful on several occasions. One example is if you want to conserve disk space.

`ntraceud -filewrap` *bytes trace_file*

The following invocation quits running **`ntraceud`**, flushes remaining trace events to the trace event file, closes the file, and removes the shared memory buffer.

`ntraceud -quit` *trace_file*

Starting Your NightTrace Application

Having already put the NightTrace library routine calls in your source code, you can now start up your application. If your application requires input, you must provide this now.

Stopping ntraceud

Once all processes in your application complete, stop the **ntraceud** daemon with a command similar to the following one:

```
ntraceud -quit trace_file
```

At this point, you can begin data analysis.

Exercise: Logging Trace Events

The following exercise has you log trace events. It is a continuation of “Exercise: Instrumenting Code” on page 3-26.

1. Compile and link **entry_exit.c** with the **ntrace** library. Give the executable the name **entry_exit**.
2. Start the **ntraceud** daemon. (Look at the `trace_start` call to determine the trace event file name.) You may need some additional options if you cannot lock pages in memory or cannot read and write to the IPL register.
3. Execute the **entry_exit** program.
4. Get the **ntraceud** daemon to give you statistics.
5. When the program completes, stop the **ntraceud** daemon.

An example solution follows.

```
$ cc -Xa -o entry_exit entry_exit.c -lntrace -lud
$ ntraceud log
$ entry_exit
$ ntraceud -stats log
$ ntraceud -quit log
```

This exercise continues in “Exercise: Displaying Trace Events” on page 5-36.

5
Invoking the ntrace Display Utility

5
Invoking the ntrace Display Utility

Overview

The trace event display utility, **`ntrace`**, is an interactive, graphical debugging and performance analysis tool. **`ntrace`** textually presents trace run statistics. As a tool built on the X Window System, it can graphically display user trace events and system trace events.

`ntrace` is flexible: you choose the look of your graphical display pages. **`ntrace`** provides many different built-in graphical components called *display objects*. You can color, select, size, position, and group these objects and direct particular trace events to specific objects; this is called *configuring* display objects. There are also ways to label trace events, trace event arguments, and other numeric values.

This chapter covers the following topics:

- X and NightTrace vocabulary
- System environment
- **`ntrace`** invocation
- **`ntrace`** options
- **`ntrace`** arguments
- **`ntrace`** user interface
- **`ntrace`** notation conventions
- **`ntrace`** Global Window

For information about textual analysis of kernel traces, see "Viewing KernelTrace Trace Event Files with ktrace" on page 11-13.

X and NightTrace Vocabulary

The Massachusetts Institute of Technology developed a windowing system called the X Window System, or X for short. If you are unfamiliar with standard X terminology, you may find the glossary near the end of this manual useful. It contains definitions of words and phrases about:

- X applications in general

- The **`ntrace`** display utility
- Window components
- Common push buttons and menu item labels
- Mouse operations

System Environment

To run the **`ntrace`** display utility, you need an installed X server. **`ntrace`** uses an X server to support windowing in trace event displays.

Motif is a user environment based on X. The window images in this manual come from a Motif environment. If you are using another environment, your windows may differ slightly from those presented here.

`ntrace` displays appear on your terminal only if you set your `DISPLAY` environment variable. Determine if this variable is set by issuing the following command:

```
$ echo $DISPLAY
```

If this variable is not set, you must set it manually to a value based on the name of your X server. For example, in Bourne shell, set the `DISPLAY` environment variable for a terminal named “eagle” this way:

```
$ DISPLAY=eagle:0.0
$ export DISPLAY
```

In the Korn shell, this is:

```
$ export DISPLAY=eagle:0.0
```

In the C shell, this is:

```
% setenv DISPLAY eagle:0.0
```

The **`.Xdefaults`** (or **`.Xresources`**) file in your login directory establishes default environmental settings for your X sessions. You may use special **`ntrace`** settings in this file to customize your **`ntrace`** displays.

`ntrace` runs on both monochrome and color monitors. See Appendix B for information about setting color and other X resources that pertain to **`ntrace`**.

TIP:

Experiment with colors and shadings until you find a set you like. To avoid visual fatigue, use highly-contrasting colors and values sparingly.

For more information on window system concepts or Motif, see “Recommended Reading” on page 1-7.

Invoking ntrace

The **ntrace** display utility resides on your system under **/usr/bin/ntrace**. It is the graphical user interface to trace event analysis. If you do not have any **ntrace**-related files but you still want to try out this tool, just type:

```
$ ntrace
```

You can override some default functionality by invoking **ntrace** with options and arguments. The full **ntrace** invocation syntax is:

```
ntrace    [-help] [-version] [-listing] [-filestats]
          [-nohardclock] [-process { all | name | PID } ]
          [-start { offset |  time{s|u} | percent% } ]
          [-end    { offset | time{s|u} | percent% } ]
          [-flat color] [-Xoption ...] [file ...]
```

Depending on your **ntrace** options and arguments, when you invoke **ntrace**, it:

- Loads <u>all</u> 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 6.)
- Displays the Global Window (See “ntrace Global Window” on page 5-26.)

The following sections discuss the **ntrace** options and arguments.

ntrace Options

You can abbreviate all **ntrace** options to their shortest unambiguous length, but most of the examples in this manual use the long option name. These options are case-insensitive. The following examples are all equivalent:

```
ntrace -help
ntrace -hel
ntrace -he
ntrace -h
ntrace -H
ntrace -HE
ntrace -Hel
ntrace -HELP
```

ntrace options include:

`-help` Displays the **`ntrace`** invocation syntax on standard output and exits. Screen 5-1 shows an example.

```
$ ntrace -help
usage:  ntrace  [-help]  [-version]  [-listing]
                 [-filestats]  [-nohardclock]  [-process {all | name | PID}]
                 [-start {offset | time{s|u} | percent%}]
                 [-end   {offset | time{s|u} | percent%}]  [-flat color]
                 [-Xoption ...]  [file ...]

Options that write to standard output:
   -help               Write this message to standard output
   -version            Write current ntrace version stamp to standard output
   -listing            Chronologically list all events to standard output
   -filestats          Write simple trace_file statistics to standard output

Options to select events:
   -nohardclock        Do not load kernel hardclock interrupt events
   -process all        Load kernel events for all user-traced processes
   -process name       Load kernel events associated with process_name
   -process PID        Load kernel events associated with PID
   -start offset       Load events after the specified event offset
   -start time{s|u}    Load events after the specified relative time
   -start percent%     Load events after the specified percent of total events
   -end offset         Load events before the specified event offset
   -end time{s|u}      Load events before the specified relative time
   -end percent%       Load events before the specified percent of total events

Options for graphical displays:
   -flat color         Color to use for all flat areas and frames
   -Xoption            Any standard X Toolkit command line options (see X(1))

Files:
   config_file         Holds configuration information:  display pages,
                       macro definitions, qualified events, qualified
                       states and tables
   event_map_file      Maps event ID numbers with event tag names
   trace_file          Holds events logged by your application and ntraceud
```

Screen 5-1. Sample Output from the ntrace -help Option

`-version` Displays the current **`ntrace`** version stamp on standard output and exits.

`-listing` Chronologically lists all trace events in the trace event file(s) to standard output and exits. The output includes the following information about a trace event: relative timestamp, trace event ID, any trace event argument(s), the global process identifier (PID) or thread name, and 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, **`ntrace`** displays symbolic trace event tags instead of numeric trace event IDs, and **`ntrace`** 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 "Understanding Event-Map Files" on page 5-10.

In kernel tracing, the **`vectors`** file provides names for system processes, interrupts, and exceptions.

Screen 5-2 and Screen 5-3 show examples from a kernel trace event file.

(Note that when viewing a user trace event file, a kernel trace event file is required in order to resolve which CPU each process was logging trace events from. See Chapter 11 for more information.)

NOTE

The information associated with the `node` field appears in this listing only when NightTrace is configured to use an RCIM to timestamp events.

```
 5536: cpu=01 TR_PAGEFLT_ADDR  pid=scheme       tid=1241'0       time=  8.305265
S user instr page fault PC=0x1000fd54
 5537: cpu=01 TR_EXCEPTION_SUS pid=scheme       tid=1241'0       time=  8.305441
S vector=inst access
 5538: cpu=01 TR_SWITCHIN      pid=idle         tid=0'0          time=  8.305441
S arg1=         0
 5539: cpu=00 TR_INTERRUPT_ENT pid=idle         tid=0'0          time=  8.313355
S vector=hardclock      level=1
 5540: cpu=00 TR_INTERRUPT_EXI pid=idle         tid=0'0          time=  8.313408
S vector=hardclock      level=0
 5541: cpu=01 TR_INTERRUPT_ENT pid=idle         tid=0'0          time=  8.313416
S vector=softclock      level=1
 5542: cpu=01 TR_INTERRUPT_EXI pid=idle         tid=0'0          time=  8.313425
S vector=softclock      level=0
```

Screen 5-2. Example of ntrace -listing Output (with instr page fault)

```
13390: cpu=01 TR_PAGEFLT_ADDR  pid=ls          tid=1250'0       time= 14.194342
S user data page fault addr=0x300ad1c0 (PC=0xb0121fbc)
13391: cpu=01 TR_EXCEPTION_EXI pid=ls          tid=1250'0       time= 14.194460
S vector=data access
13392: cpu=01 TR_SYSCALL_ENTRY pid=ls          tid=1250'0       time= 14.194473
S syscall=read  device=file
13393: cpu=01 TR_EXCEPTION_ENT pid=ls          tid=1250'0       time= 14.194528
S vector=data access
13394: cpu=01 TR_PAGEFLT_ADDR  pid=ls          tid=1250'0       time= 14.194534
S kernel data page fault addr=0xe1e18000 (PC=0x000931cc)
13395: cpu=01 TR_EXCEPTION_EXI pid=ls          tid=1250'0       time= 14.194590
S vector=data access
13396: cpu=01 TR_SYSCALL_EXIT  pid=ls          tid=1250'0       time= 14.194659
S syscall=read  device=file
13397: cpu=01 TR_SYSCALL_ENTRY pid=ls          tid=1250'0       time= 14.194715
S syscall=close device=file
```

Screen 5-3. Example of ntrace -listing Output (with data page fault)

-filestats Displays simple statistics about all trace event file(s) arguments to standard output, similar to the display on the Global Window, and exits. (See "ntrace Global Window" on page 5-26.) 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, and the number of trace events lost.

Screen 5-4 shows an example, with:

log	The user trace event file.
map	The event-map file.
continuation events	The NT_CONTINUE trace events that **ntraceud** logs for multi-argument trace events.

```
amber2> ntrace -filestats n1.cap vectors.cap | p

1 trace event log file read.

Kernel trace event log file:  n1.cap.
     10916 trace events plus 9863 continuation events.
     10916 events saved in memory.
     0 trace events lost.
     52.4036288s time span, from 0.0000000s to 52.4036288s.

RCIM synchronized tick clock was used to time stamp events.

10916 total events read from disk plus 9863 continuation events.
10916 total events saved in memory; 1 events internal to ntrace.
0 total trace events lost.
52.4036288s total time span saved in memory.
```

Screen 5-4. Example of ntrace -filestats Output

By default, when **`ntrace`** starts up, it reads and loads all trace events from all trace event files into memory; therefore, the more trace events in your trace event file(s), the more memory **`ntrace`** uses. The **`-nohardclock`**, **`-process`**, **`-start`**, and **`-end`** options let you prevent the loading, but not the reading, of certain trace events.

`-nohardclock`	Do not load hardclock interrupts from the kernel trace event file. This option may save about 15% of the memory **`ntrace`** consumes. For more information on the hardclock interrupt, see "Hardclock Interrupt Handling" in the *PowerMAX OS Real-Time Guide*.

If you invoke **`ntrace`** with the **`-process`** option, it loads only exceptions and system calls of processes you specify after the **`-process`**; this takes some extra processing time during **`ntrace`** start up. You can invoke **`ntrace`** with multiple **`-process`** options. The possible ways to use the **`-process`** option include:

`-process all`	From the NightTrace kernel trace event file, load only exceptions and system calls associated with process(es) in the user trace event file(s). This implies that you invoke **`ntrace`** with both a kernel trace event file and user trace event file(s).
`-process` *PID*	From the NightTrace kernel trace event file, load only exceptions and system calls associated with this global process identifier (*PID*). Note that a global PID is different than a raw PID. For more information on global process identifiers see "PID List" on page 8-7.
`-process` *name*	From the NightTrace kernel trace event file, load only exceptions and system calls associated with this process name, *name*. This implies that you invoke **`ntrace`** with both a kernel trace event file and user trace event file(s).
`-start` *offset*	Load trace events after the specified trace event offset. (See "The Grid" on page 6-4 for information about trace event offsets.)
`-start` *time*{**`s`**|**`u`**}	Load trace events after the specified relative time in seconds (`s`) or microseconds (`u`).
`-start` *percent***`%`**	Load trace events after the specified percent of total trace events. The `%` is required.
`-end` *offset*	Load trace events before the specified trace event offset.
`-end` *time*{**`s`**|**`u`**}	Load trace events before the specified relative time in seconds (`s`) or microseconds (`u`).
`-end` *percent***`%`**	Load trace events before the specified percent of total trace events. The `%` is required.

For example, the following invocation loads trace events logged between 5 and 15 seconds into the trace session.

```
$ ntrace -start 5s -end 15s log
```

For example, the following invocation skips the first 10% of trace events, loads the next 15% of trace events, and skips the remaining 75% of trace events.

```
$ ntrace -start 10% -end 25% ulog
```

If you invoke **`ntrace`** with several **`-start`** options, **`ntrace`** pays attention only to the last one. The same is true if you invoke **`ntrace`** with several **`-end`** options. If you invoke **`ntrace`** with both a **`-start`** and a **`-end`** option and the **`-end`** condition precedes the **`-start`** condition, **`ntrace`** does not load any real trace events; it loads two dummy trace events.

You can establish a default windowing environment for all your **`ntrace`** sessions in your **`.Xdefaults`** (or **`.Xresources`**) file. You can invoke **`ntrace`** with X options to:

- Customize an individual **`ntrace`** session
- Override any corresponding settings in the **`.Xdefaults`** file
- Possibly improve the readability of your **`ntrace`** display

You can invoke **`ntrace`** with the following options:

`-flat` *color*	Color to use for the window edges, scroll bars, push buttons, and menu bars.
`-Xoption`	This option includes all of the standard X Tool kit command-line options (see **`X(1)`**).

TIP:
Consider experimenting with these options and then saving their counterpart values in your **`.Xdefaults`** or **`.Xresources`** file.

Invoking **`ntrace`** on a color X server with no **`ntrace`** options and no **`ntrace`** settings in **`.Xdefaults`** is nearly equivalent to:

```
$ ntrace -fg black -bg white -flat gray75 -fn fixed
```

Your X terminal vendor supplies you with vendor-specific directories and files that pertain to colors and fonts. The file that contains available colors is called **`rgb.txt`**. The directory for fonts is **`/usr/lib/X11/fonts`**. For more information on X options, see **`xterm(1)`** or **`X(1)`**.

ntrace Arguments

You can invoke **`ntrace`** with arguments that provide information about trace events, their tags, other labels, and desired display object layout. **`ntrace`** identifies the purpose of a file argument by its contents; therefore, the order (and number) of these arguments is not significant.

SYNTAX

ntrace [-*option*] [*trace_files*] [*event_map_files*] [*config_files*] **vectors**

ARGUMENTS

trace_files	Trace event files contain sequences of trace events that your application and the **ntraceud** daemon logged or NightTrace kernel trace events logged by the kernel trace program, **ktrace(1)**, and converted by **ntfilter(1)**.
event_map_files	Event-map files map short mnemonic trace event tags to numeric trace event IDs and associates data types with trace event arguments. This is a hand-edited ASCII file.
config_files	Configuration files define macros, qualified events, qualified states, string tables, format tables, display objects, and display pages. These ASCII files are usually created with **ntrace**.
vectors	The **vectors** file contains definitions of the `vector`, `syscall`, and `pid` string tables that provide names for system processes, interrupts, and exceptions that occurred during kernel tracing. The **ntfilter** tool dynamically generates this file for kernel-trace analysis. See “Converting KernelTrace Trace Event Files with ntfilter” on page 11-21 for details.

See the *NightTrace Pocket Reference* card for a syntax summary of formats for event-map files, string tables, and format tables.

Understanding Trace Event Files

You can invoke **ntrace** without any trace event file names as arguments, but you cannot examine any trace events this way. Therefore, this is rarely done. Normally you invoke **ntrace** with one or more user trace event files and/or a kernel trace event file. Invoking **ntrace** with multiple trace event files is mainly useful when you have run several simultaneous, related trace sessions and you wish to merge their trace events into a single display.

In user tracing, **ntraceud** creates a trace event file to hold your application’s trace events and some of NightTrace’s own trace events. Invoking the **ntraceud** daemon with the **-quit** option causes it to flush and close the trace event file. In kernel tracing, the **ktrace** tool creates a KernelTrace trace event file to hold kernel trace events. Invoking the **ntfilter** tool with this KernelTrace trace event file causes it to write the trace events into a trace event file with a format compatible with **ntrace**. Once this is done, you can invoke **ntrace** with the trace event file names as its argument .**ntrace** reads the trace event files, puts information about all loaded trace events in memory, and displays the trace events chronologically in the layouts you choose.

You can create NightTrace kernel trace event files with the **ktrace(1)** and **ntfilter(1)** tools. See “Kernel Tracing with ktrace” on page 11-8 and “Converting KernelTrace Trace Event Files with ntfilter” on page 11-21 for details.

Because of the overhead involved in loading trace events into memory, **`ntrace`** loads trace events only at start up. **`ntrace`** never prompts you for a trace event file name.

TIP:

Invoke **`ntrace`** only with the trace events and trace event files you need. Use options to **`ntrace`** to limit which trace events get loaded into memory. The more trace events **`ntrace`** must process, the slower its start up and display updates. For more information, see "Conserving Memory and Accelerating ntrace" on page A-4.

A trace event file consists of:

- One file header record
- Several trace event and continuation records

The header record contains some NightTrace statistics that pertain to the whole trace session. The trace event records describe individual trace events logged with zero or one numeric argument; these records may come from your application, from NightTrace itself, or in kernel tracing from the kernel. Continuation records (with trace event tag `NT_CONTINUE`) describe any other arguments logged with the trace event. The exact format of the trace event file appears on the **`ntrace(4)`** man page.

Although you must have read permission to a trace event file, there is no reason to edit and rarely a reason to examine the contents of this file; however, you can inspect it with the **`od(1)`** octal dump command. Because trace event files begin with a specific magic number, running the **`file(1)`** command on a trace event file named **`log`** has the following result:

```
$ file log
log:    NightTrace trace event file
```

If **`ktrace(1)`** creates a KernelTrace trace event file that you name **`klog`**, then running **`file(1)`** on this file has the following result:

```
$ file klog
klog:   KernelTrace trace event file
```

Understanding Event-Map Files

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

A *trace point* is a location in the application's source code where you call a NightTrace trace event logging routine. Each trace point has a corresponding trace event ID number. An *event-map file* allows you to associate meaningful tags or labels 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 **`ntrace`** does not require you to use event-map files, labels and correct display formats can make graphical **`ntrace`** displays and textual summary information much more readable.

You can run **`ntrace`** with multiple event-map files; however, if a trace event ID is multiply-defined, **`ntrace`** writes an error message in the message display area of the

Global Window. For more information on the Global Window, see “ntrace Global Window” on page 5-26.

TIP:

If you used symbolic constants to represent numeric trace event IDs, you may be able to simply reformat this information in the event-map file.

To load an existing event-map file, either:

- Start an **`ntrace`** session with the file name as an argument
- Click on File Ì Read Event-Map File ... on the Global Window

You must create an event-map file with a text editor before you invoke **`ntrace`**. The file contains lines of ASCII text separated by newlines. There is one trace event tag mapping per line. White space separates each field except the conversion specifications; commas separate the conversion specifications. **`ntrace`** 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_tag”* `[` *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_tag	A character string to be associated with *event_ID*. Trace event tags must begin with a letter and consist solely of alphanumeric characters and underscores.Keep trace event tags short; otherwise, **`ntrace`** may be unable to display them in the limited window space available.
	The following words are reserved in **`ntrace`** and should not be used in upper case or lower case as trace event tags: `NONE, ALL, ALLUSER, ALLKERNEL, TRUE, FALSE, CALC`.

TIP:

Consider giving your trace events upper case tags in event-map files and giving any corresponding qualified event the same name in lower case. For more information about qualified events, see “Qualified Events” on page 9-81.

If your application logs a trace event with one or more numeric arguments, by default **`ntrace`** 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 **`ntrace`** with the event-map file and the **`-listing`** option, **`ntrace`** shows only *nargs* arguments for the trace event.

conv_spec	A conversion specification or display format for a trace event argument. **`ntrace`** uses conversion specification(s) to display the trace event's argument(s) in the designated format(s). There must be one conversion specification per argument. Valid conversion specifications for displays include the following:

`%d`	signed decimal integer (default)
`%o`	unsigned octal integer
`%x`	unsigned hexadecimal integer
`%lf`	signed double precision, decimal floating point

For more information on these conversion specifications, see **`printf(3S)`**.

The following line is an example of an entry in an event-map file:

```
event: 5 “Error” 2 %x %lf
```

Trace event ID `5` is an error condition; when appropriate, **`ntrace`** displays trace event 5 and labels the trace event `“Error.”` Trace event 5 also has two (`2`) arguments. **`ntrace`** displays the first argument in unsigned hexadecimal integer (`%x`) format and the second argument in signed double precision decimal floating point (`%lf`) format. (You may override these conversion specifications when you configure display objects.)

For more information on event-map files, see“Pre-Defined String Tables” on page 5-15, “Read Event-Map File” on page 5-32, and the **`ntrace(4)`** man page. For information about trace event tags for kernel trace events, see the **`/usr/lib/NightTrace/eventmap`** file

Understanding Page Configuration Files

`ntrace` does not require you to use configuration files. However, using these file(s):

- Allows you to associate macros, qualified events, and qualified states with particular display page(s)
- Improves the readability of your displays
- Saves you time laying out your display pages

A *configuration file* is an ASCII file that contains definitions. These definitions look like initialized C structures. A configuration file can contain any number of the following definitions:

- Macro, qualified event, and qualified state definitions (See Chapter 9.)
- String table definitions (See “String Tables” on page 5-14.)
- Format table definitions (See “Format Tables” on page 5-18.)
- Display page definitions

NOTE

The components of a configuration file are often interrelated. For example, display pages may reference user-defined tables. **`ntrace`** generates an error message if your configuration file refers to a table before you define it. To avoid this problem, create your configuration files so that a table definition precedes its reference(s). There is no similar problem for macros, qualified events, and qualified states.

If you accidentally define a macro, qualified event, or qualified state more than once in a configuration file, **`ntrace`** flags subsequent definitions as errors. If you define a string table or format table more than once in a configuration file, **`ntrace`** merges the two tables; if there are duplicate entries, values come from the last definition.

Results can be unpredictable if multiple users simultaneously modify a configuration file. Similarly, **`ntrace`** may behave strangely if you edit your configuration file so that any of your display objects overlap.

You can create, modify, save, and load configuration files from within **`ntrace`**; however, you must use a text editor to create and modify tables in a configuration file. **`ntrace`** 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, do one of the following:

- Start an **`ntrace`** session with the file name as an argument
- Click on File Ì Open Config File … on the Global Window

For more information on manipulating configuration files, see “File Menu Item” on page 5-27.

ntrace Tables

The configuration file may contain two types of tables: string tables and format tables. Both types of table can improve the readability of your **`ntrace`** displays. The only way to put tables into your configuration file is by text editing the file before you invoke **`ntrace`**. To avoid any forward-reference problems, define all string tables before any format tables.

A table lets you associate meaningful character strings with the more cryptic integer values, such as trace event arguments. These character strings may appear in **`ntrace`** displays.

For example, suppose that one trace event’s argument may have the decimal values 2, 5, and 8. You can define:

- A string that appears when the value is 2

- A different string that appears when the value is 5
- A third string that appears when the value is 8

The following table names are reserved in **`ntrace`** and should not be redefined in upper case or lower case:

> `event`, `pid`, `tid`, `boolean`, `name_pid`, `name_tid`, `node_name`, `pid_`*nodename*, `tid_`*nodename*, `vector`, `syscall`, `device`, `vector_`*nodename*, `syscall_`*nodename*, `device_`*nodename*, `event_summary`, `event_arg_summary`, `event_arg_dbl_summary`, `state_summary`

The results are undefined if you supply your own version of these tables. For more information on pre-defined tables, see "Pre-Defined String Tables" on page 5-15, "Pre-Defined Format Tables" on page 5-21, and "Kernel String Tables" on page 11-32.

TIP:

Put tables in separate configuration files from display pages. This way tables do not get redefined if you close and reopen a display page during a single **`ntrace`** session.

If you define a string table or format table more than once in a configuration file, **`ntrace`** 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" ; ]
};
```

You do not need to separate the parts (tokens) of the string table with white space. String-table tokens are indivisible; although these tokens need not break on the lines shown, they must appear in the order shown. 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 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, **`ntrace`** 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 with a single string. The fields of the default item line are:

`default_item`	The keyword that begins all default item lines.
str_const	(See *str_const* above.)

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.

`ntrace` 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 **`ntrace`** cannot find a particular string table **`ntrace`** 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 **`ntrace`** 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 “get_string()” on page 9-75 and the **`/usr/lib/NightTrace/tables`** file.

Pre-Defined String Tables

The following string tables are pre-defined in **`ntrace`**:

`event`	A dynamically generated string table internal to **`ntrace`**. It maps all known numeric trace event IDs with symbolic trace event tags. A similar association appears in the **`/usr/lib/NightTrace/eventmap`** file; this is an

event-map file that associates trace event IDs with kernel trace event tags.

This table is indexed by an event code or an event code name. Examples of using this table are:

```
get_string(event, 4112)
get_item(event, “TR_INTERRUPT_EXIT”)
```

`pid`

A dynamically generated string table internal to **`ntrace`**. 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.

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

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

```
get_string(pid, pid())
get item(pid, “ntraceud”)
```

See “PID List” on page 8-7 for more information.

`tid`

A dynamically generated string table internal to **`ntrace`**. In user tracing, it associates NightTrace thread ID numbers with thread names. In kernel tracing, this table is not used.

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

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

```
get_string(tid, tid())
get_item(tid, “cleanup_thread”)
```

See “TID List” on page 8-8 for more information.

`boolean`	A string table defined in the **`/usr/lib/NightTrace/tables`** file. It associates `0` with `false` and all other values with `true`.

`name_pid`	A dynamically generated string table internal to **`ntrace`**. It maps all known node ID numbers (which are internally assigned by **`ntrace`**) 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”)`

`name_tid`	A dynamically generated string table internal to **`ntrace`**. It maps all known node ID numbers (which are internally assigned by **`ntrace`**) 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 **`ntrace`**. It associates node ID numbers (which are internally assigned by **`ntrace`**) 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 **`ntrace`**. 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 **`ntrace`**. 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 “Kernel String Tables” on page 11-32.
`syscall`	See “Kernel String Tables” on page 11-32.
`device`	See “Kernel String Tables” on page 11-32.
`vector_`*nodename*	See “Kernel String Tables” on page 11-32.
`syscall_`*nodename*	See “Kernel String Tables” on page 11-32.
`device_`*nodename*	See “Kernel String Tables” on page 11-32.

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 “get_string()” on page 9-75. For examples of function calls with these tables, see Table 8-3.

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” ] ... ; ]
};
```

You do not need to separate the parts (tokens) of the format table with white space. Format table tokens are indivisible; although these tokens need not break on the lines shown, they must appear in the order shown. 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 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, **`ntrace`** 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(3S)`**.

value1 A value associated with the first conversion specification in *format_string*. The value may be a constant string (literal) expression or an **`ntrace`** expression. A string literal expression must begin and end with a `\\'` and must be enclosed in double quotes; for example:

```
“\\’string expression\\’"
```

An expression may be a `get_string()` call; a description of the `get_string()` function appears in “get_string()” on page 9-75. For more information on expressions, see Chapter 9.

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 decimal integer. If the number or data type of the quoted value(s) do not match *format_string*, the results are not defined.

The optional *default item line* associates all other integer values with a single format item. **`ntrace`** 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. The fields of the default item line are:

`default_item`	The keyword that begins all default item lines.
format_string	(See *format_string* above.)
value1	(See *value1* above.)

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 **`ntrace`** display:

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

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

```
The current state is Initializing
```

When trace event `999` occurs, **`ntrace`** 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 **`ntrace`** displays:

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

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

For more information on `get_string()`, see “get_string()” on page 9-75. For more information on format tables and the `get_format()` function, see “get_format()” on page 9-79. See also the **`/usr/lib/NightTrace/tables`** file. For more information about `arg1()`, see “arg()” on page 9-20. For more information about `offset()`, see “offset()” on page 9-27.

Pre-Defined Format Tables

The following format tables are pre-defined in the **`/usr/lib/NightTrace/tables`** file:

`state_summary`	Formats statistics about the state matches summarized, state durations, and state time gaps. This table provides the default state summary output format.
`event_summary`	Formats statistics about the trace event matches and trace event time gaps. This table provides the default trace event summary output format.
`event_arg_summary`	Formats statistics about the trace event matches and their type long trace event arguments.
`event_arg_dbl_summary`	Formats statistics about the trace event matches and their type double trace event arguments.

For more information about summaries, see Chapter 10.

You can use pre-defined format tables anywhere that format tables are appropriate. Use the `get_format()` function to look up values in format tables. For information about the `get_format()` function, see “get_format()” on page 9-79. For examples of function calls with format tables, see Table 8-3.

Configuring Display Pages

The configuration file usually contains display page(s). You use **`ntrace`** to put these display page(s) in your configuration file.

`ntrace` lets you customize the layout of your **`ntrace`** display pages. You do this by coloring, selecting, sizing, positioning, grouping, and otherwise configuring display objects on a particular display page. Once you have created a useful display page, you may save it for future **`ntrace`** sessions. Saving a display page is the same as creating or modifying a configuration file. See Chapter 7 and Chapter 8 for more information.

Rather than creating your own display page, you can let **`ntrace`** create a default display page for you. For more information on the default display page, see "Default Page" on page 5-29 and "Kernel String Tables" on page 11-32.

ntrace User Interface

`ntrace` displays textual and graphical information, and it provides you with ways to manipulate this information. These displays and mechanisms make up the **`ntrace`** user interface.

The next sections describe the following **`ntrace`** user interface issues:

- Using the mouse
- Understanding pointer shapes
- Anticipating window layout
- Resizing windows

Using the Mouse

It is assumed that your X server has a three-button mouse. By default, mouse button 1 is the leftmost button, button 2 the middle button, and button 3 the rightmost button. You can reassign the functions associated with mouse buttons by using **`xmodmap(1)`**. If you do not have a three-button mouse or a standard 101-key North American keyboard, see your system administrator or read sections on input and navigation in the *OSF/Motif Style Guide*.

You use the mouse with point-and-click interfaces in **`ntrace`**. Each mouse button has a different purpose. The only mouse button operation you need to know for now is that clicking mouse button 1 usually does a single selection.

Understanding Pointer Shapes

When you move the mouse, the *mouse pointer* moves on the screen. You use this pointer to point to particular parts of the screen. The shape of the pointer shows the current usage. The following table describes when **`ntrace`** uses each pointer shape.

Table 5-1. ntrace Pointer Shapes and Their Usage

Shape	When Used
↖	By default
↗	When a menu action is pending
✥	While moving a display object
⇱ ⤒ ⇗ ⇤ ⇥ ⇙ ⤓ ⇲	While resizing a display object
+	During display-object placement
⌚	During time-consuming operations, for example while scrolling through a large trace event file

Anticipating Window Layout

Your *window manager* may automatically place a window frame around your windows. It may also provide you with a means of performing some standard operations, such as minimizing (also known as *iconifying*) or maximizing the window size. If your window manager provides a window frame, then **`ntrace`** puts a window title in the title area of this frame. If you minimize a window, **`ntrace`** provides all or part of the window title for the icon that represents that window.

`ntrace` windows may contain different mixtures of components. In **`ntrace`** the window components include:

Menu bars and pull-down menus
: A *menu bar* appears at the top of the window. It consists of labeled *pull-down menu*(s). When you *select* (click on) a pull-down menu, *entries* appear in a vertical list. Selecting a menu entry with ellipses on it, causes a *dialog box* or form to appear. Entries are grouped by function with *separators* dividing groups. Any destructive items appear last.

Push buttons
: A *push button* is a graphic image of a labeled button on a *panel*. Push buttons are evenly-spaced in a horizontal panel at the bottom of the window. The default push button is highlighted by having a border around it. Pressing `<Enter>` makes the default push button take effect. Push buttons are organized by frequency of use and grouped by functionality. Push button names are active verbs. The most-commonly-used push buttons in **`ntrace`** are: `Apply`, `Reset`, and `Close`.

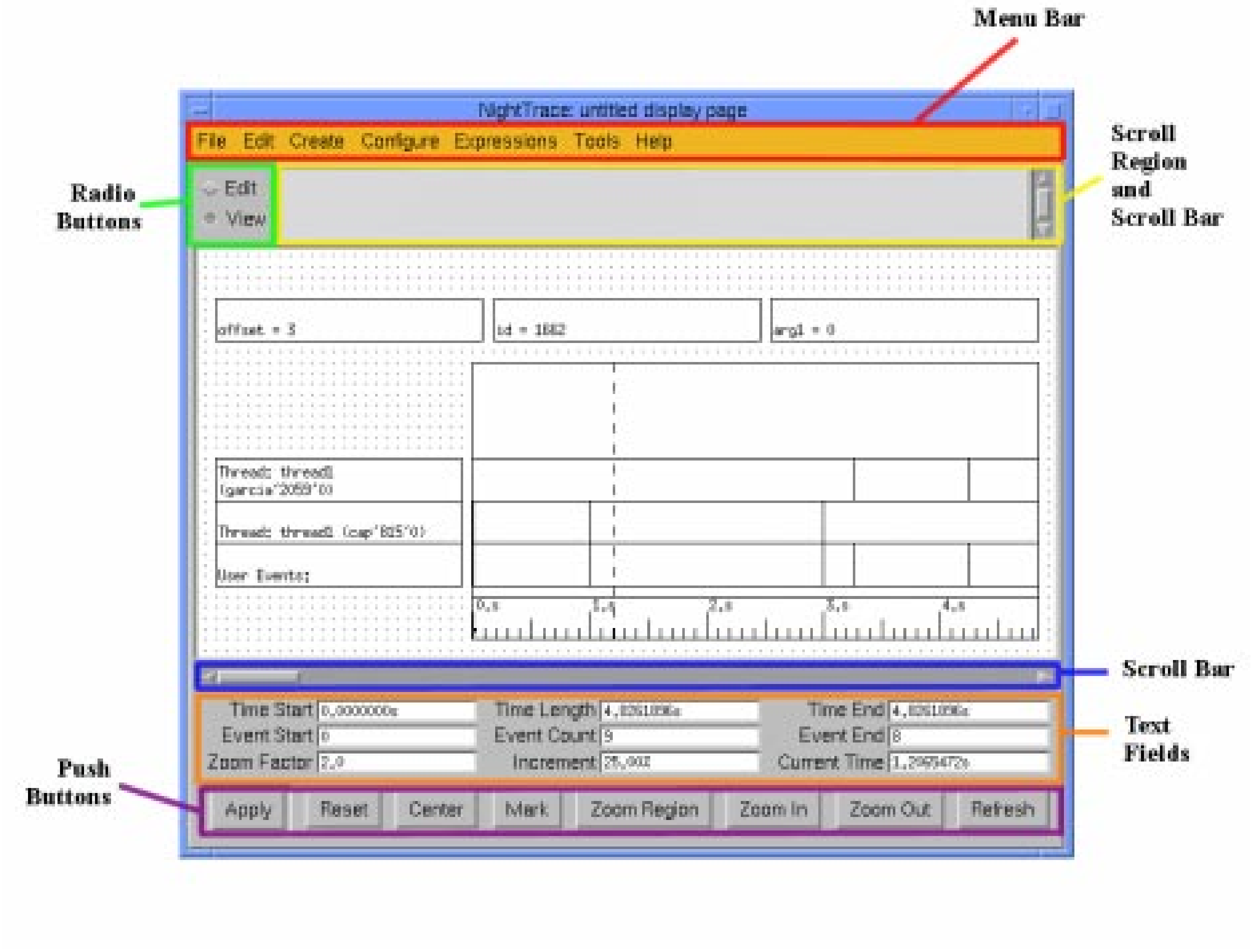
Rather than duplicating functionality in both a menu entry and push button, **`ntrace`** supports the menu entry.

Radio buttons
: A *radio button* is a graphic, labeled diamond-shape that represents a mutually exclusive selection from related radio buttons. Related radio buttons usually appear on the same panel, a panel between the menu bar and the push buttons. The first radio button is the default one. The selected radio button has a different color than the others.

Scroll regions and scroll bars
: A *scroll region* appears between the menu bar and the push buttons. The *scroll bar* is immediately below or to the right of the scroll region.

Text fields
: A *text field* appears between the menu bar and the push buttons. While a text field is being edited, it contains a blinking vertical bar called the *text cursor*. The text cursor shows your current edit position within the field.

Figure 5-1 illustrates these window components.


Menu Bar
Scroll Region and Scroll Bar
Radio Buttons
Scroll Bar
Text Fields
Push Buttons
NightTrace: untitled display page
File Edit Create Configure Expressions Tools Help
Edit
View
Apply Reset Center Mark Zoom Region Zoom In Zoom Out Refresh

offset = 3   id = 1682   arg1 = 0

Thread: thread1 (garcia'2059'0)
Thread: thread2 (cap'825'0)
User Events:

Time Start	0.0000000s	Time Length	4.8261896s	Time End	4.8261896s
Event Start	0	Event Count	9	Event End	8
Zoom Factor	2.0	Increment	25.00%	Current Time	1.2985472s

Figure 5-1. Window Components

Resizing Windows

You can resize all windows in **`ntrace`**. However, many windows require a minimum size to display information. You can resize some windows only horizontally. When you make a window wider, **`ntrace`** horizontally stretches any push buttons in that window to take up the new width.

ntrace Notation Conventions

This manual uses the following notation convention to reference menu entries.

Menu bars consist of one or more *menu items*. Clicking on a menu item causes a pull-down menu to appear. *Pull-down menus* have selectable *entries*. This manual lists menu levels (from menu item to menu entry) in the order in which they appear. An arrow

(Ì) separates each menu level from the next. To show the progression from menu item to menu entry, this manual uses the following notation:

menu item Ì menu entry

For example, if you click on the File menu bar item, you may then select the Exit menu entry. This manual shows this procedure as:

Click on File Ì Exit

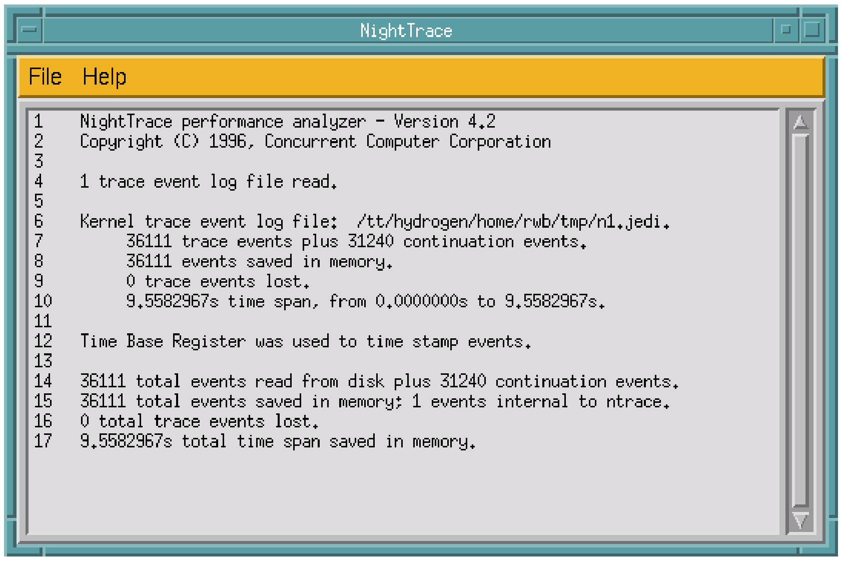
If the menu item consists of more than one word, the procedure is shown as:

Click on File Ì Open Config File …

ntrace Global Window

Unless you invoke **`ntrace`** with an option that writes to standard output, **`ntrace`** starts up by displaying the Global Window. The **`ntrace`** Global Window consists of:

- A message display area and its associated scroll bar
- A menu bar



```
NightTrace performance analyzer - Version 4.2
Copyright (C) 1996, Concurrent Computer Corporation

1 trace event log file read.

Kernel trace event log file:  /tt/hydrogen/home/rwb/tmp/n1.jedi.
     36111 trace events plus 31240 continuation events.
     36111 events saved in memory.
     0 trace events lost.
     9.5582967s time span, from 0.0000000s to 9.5582967s.

Time Base Register was used to time stamp events.

36111 total events read from disk plus 31240 continuation events.
36111 total events saved in memory; 1 events internal to ntrace.
0 total trace events lost.
9.5582967s total time span saved in memory.
```

Figure 5-2. Global Window for a Single Trace Event File

Message Display Area

The message display area of the Global Window presents the following information:

- Version information

- Trace-session statistics grouped by trace event file
- Cumulative statistics for all trace event files
- Event timestamp source
- Error messages about corrupted or syntactically-incorrect **ntrace** file arguments

You can also obtain this statistical information by invoking **ntrace** with the **-filestats** option. If you invoke **ntrace** without any trace event file arguments, **ntrace** displays most of the statistics as zero values.

By default, **ntrace** is sized to hold twenty lines in the message display area. You can alter this number by changing the size of the Global Window. To change the Global Window size, resize your window by using features of your window manager. It is not necessary to resize a window to see messages 21 and higher; you can scroll through all messages by using the scroll bar.

Menu Bar

The menu bar of the Global Window consists of the following menu items:

- File
- Help

File Menu Item

When you click on the File menu item on the Global Window, the pull-down menu shown in Figure 5-3 appears.

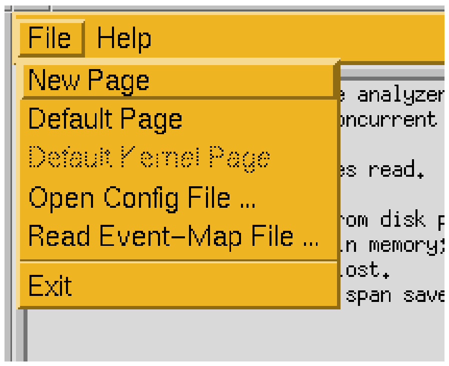


Figure 5-3. Global Window File Menu

New Page, Default Page, Default Kernel Page, and Open Config File ..., can all cause **ntrace** to bring up a display page. The difference between these menu items is the origin and content of the display page. The Default Kernel Page is disabled (dimmed) unless you provide a NightTrace kernel trace file. It is discussed in "Kernel Display Pages" on page 11-22. The rest of these menu entries are discussed later in this chapter. The regular features of a display page include:

- Create and configure display objects so they graphically depict your trace session (See Chapter 7 and Chapter 8)
- Examine trace events, states, trace event arguments, and timings using different display objects (See Chapter 6)
- Create, configure, and modify macros, qualified events, and qualified states (See Chapter 9)
- Search for trace events (See Chapter 10)
- Summarize data into statistics for trace events and states (See Chapter 10)

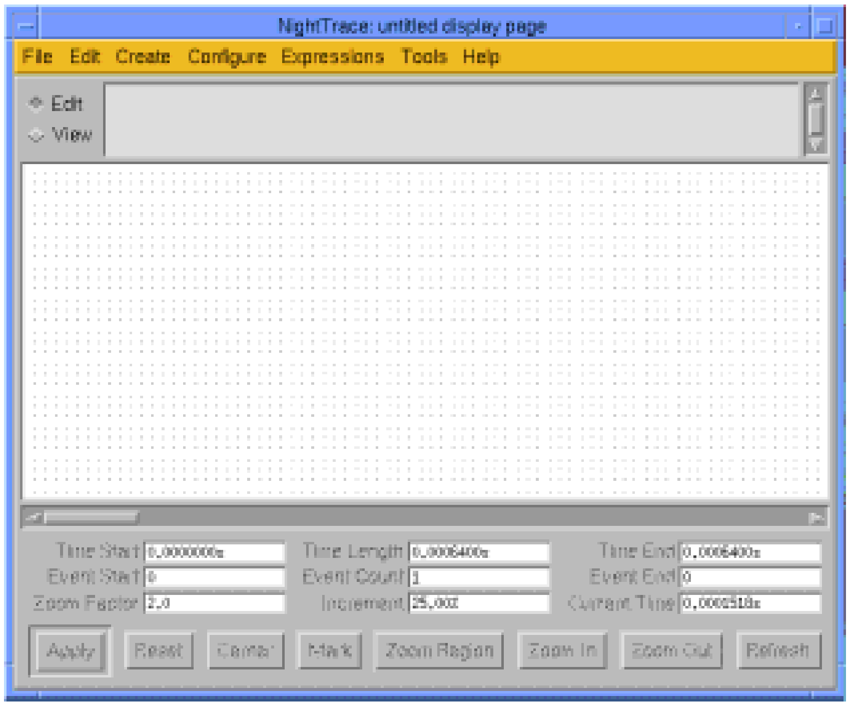
New Page

When you click on File Ì New Page on the Global Window, a display page with a dotted but otherwise empty grid appears. This menu item is useful if you want to create a display page from scratch. Most regular features of a display page are immediately available on a new page. However, you must create and configure display objects before you can:

- Examine trace events, states, and trace event arguments using different display objects
- Graphically depict your trace session

NOTE

The new display page comes up in Edit mode. In Edit mode, the field labels, scroll bar and push buttons on the display page are disabled (dimmed).


NightTrace: untitled display page
File Edit Create Configure Expressions Tools Help
Edit
View
Apply
Reset
Center
Mark
Zoom Region
Zoom In
Zoom Out
Refresh

Time Start	0.0000000s	Time Length	0.0006400s	Time End	0.0006400s
Event Start	0	Event Count	1	Event End	0
Zoom Factor	2.0	Increment	25.00%	Current Time	0.0002518s

Figure 5-4. New Display Page

Default Page

When you click on `File` Ì `Default Page` on the `Global Window`, an automatically-generated display page appears. The purpose of the default display page is to save you time configuring display objects; if this page does not exactly meet your needs, you can customize it or you can create a new display page. **`ntrace`** brings up this page in View mode. All regular features of a display page are immediately available on a default page.

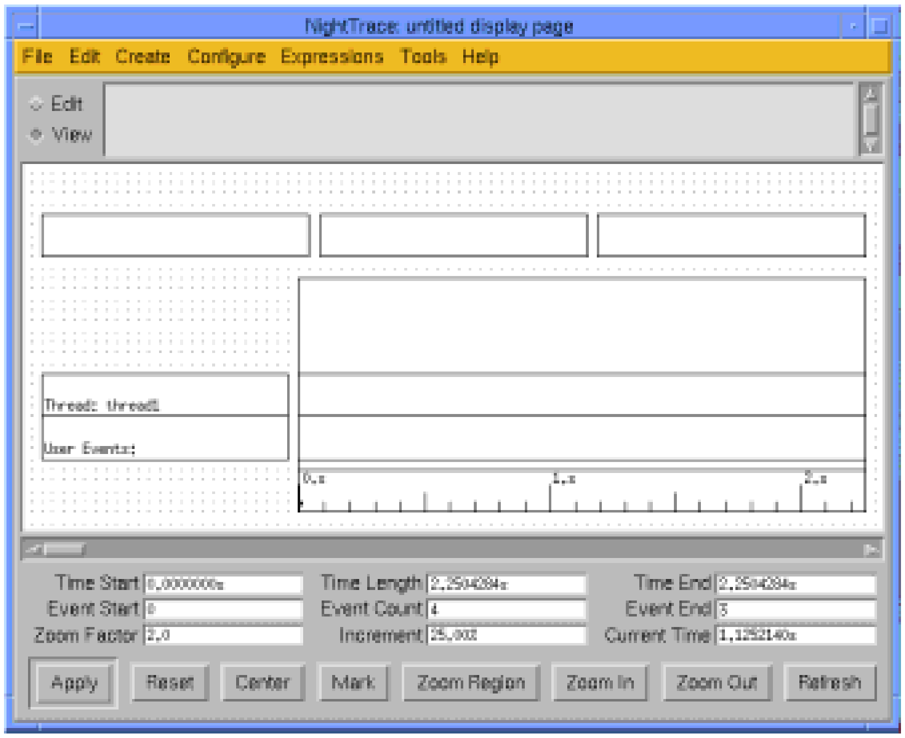
Before **`ntrace`** displays a default display page, it counts the number of processes in your trace event file(s). Normally, it then creates and displays a default display page with one StateGraph per process in the trace event file(s). The StateGraph is configured to show all user trace events. If there are so many processes that their StateGraphs do not fit on the screen, then **`ntrace`** does not display any StateGraphs.

When analyzing trace events timestamped by an RCIM synchronized tick clock, if a thread name is not unique in the trace events, **`ntrace`** prints a node name along with the process ID number and thread ID number in the associated GridLabel to identify that thread.

Figure 5-5 shows a default display page. A brief description of the display objects on this display page follows. See Chapter 7 for a more detailed description of display objects and the display page.

NOTE

The dynamic information that display objects present relates to the interval, a window into your trace session.



Time Start	0.0000000s	Time Length	2.2504284s	Time End	2.2504284s
Event Start	0	Event Count	4	Event End	3
Zoom Factor	2.0	Increment	25.00%	Current Time	1.1252140s

Figure 5-5. A Default Display Page

GridLabel	A textual display object that contains a static user-specified text string. This object labels other objects for clarity
DataBox	A display object that displays textual or numeric information such as the trace event ID or tag and the time the trace event occurred. The information it displays is related to the current time. A DataBox's main use is to display data that is variable in nature and does not lend itself to graphical representation.
Column	A scrollable display object that does not display data itself but holds graphical display objects: StateGraphs, EventGraphs, DataGraphs, and Rulers. Its purpose is to define the width of and group together graphical display objects.
StateGraph	A scrollable display object that graphically displays states as bars and trace events as vertical lines in a Column. The StateGraph shows relative chronological positions of trace events and states since the trace started.

EventGraph	A scrollable display object that graphically displays trace events as vertical lines in a Column. It shows relative chronological positions of trace events since the trace started.
DataGraph	A scrollable display object that graphically displays a trace event expression as a vertical line or bar in a Column. A DataGraph shows the relative chronological positions of trace event arguments since the trace started. The height of the line or bar is proportional to the value of the expression.
Ruler	A scrollable display object resembling a ruler that graphically displays the time. A Ruler appears within a Column and shows what time a trace event occurred in a StateGraph, EventGraph, or DataGraph.

Open Config File

A configuration file contains user-created display page(s), macros, qualified events, qualified states, and/or table definitions. When you click on File Ì Open Config File … on the Global Window, a dialog box appears. This window prompts you for an existing configuration file name. To avoid any **ntrace** errors, you must have read permission for the file. When you open a configuration file, all regular features of a display page are immediately available.

When you open an existing configuration file that contains display page(s), **ntrace** displays them. This can save time configuring display objects and let you immediately begin trace event analysis.

The Open Config File Dialog Box (shown in Figure 5-6) is made up of:

- A File Name text field
- A Filter text field
- A file name scroll region and its scroll bar
- The (default) Open push button
- The Cancel push button

If you decide not to open a configuration file, click on Cancel. This action causes the Open Config File Dialog Box to go away. If you still want to open an existing configuration file, see “The File Selection Dialog Box” on page 5-34.

For more information on configuration files, see “Understanding Page Configuration Files” on page 5-12.

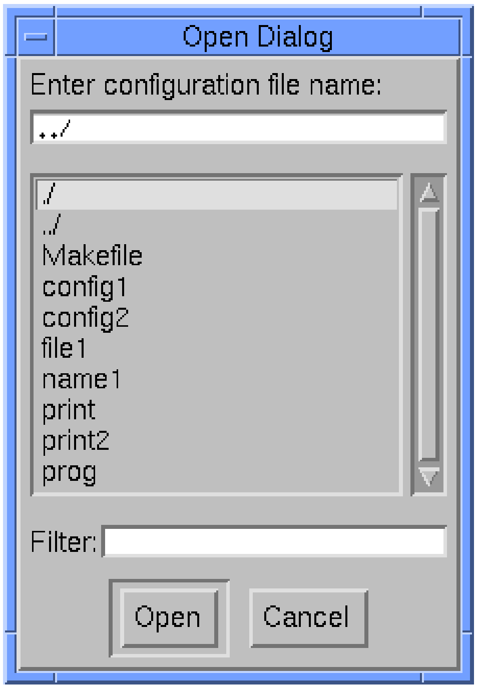


Figure 5-6. The Open Config File Dialog Box

Read Event-Map File

An event-map file associates user-created mnemonic tags or labels with numeric trace event IDs. When you click on File Ì Read Event-Map File … on the Global Window, a dialog box appears. This window prompts you for an existing event-map file. To avoid any **ntrace** errors, you must have read permission for this file.

Once **ntrace** reads an event-map file, it is able to display short trace event tags rather than less-meaningful trace event IDs. By naming your trace events, you can make your displays much easier to understand.

The Read Event-Map File Dialog Box is made up of:

- A File Name text field
- A Filter text field
- A file name scroll region and its scroll bar
- The (default) Read push button
- The Cancel push button

If you decide not to read an event-map file, click on Cancel. This action causes the Read Event-Map File Dialog Box to go away. If you still want to read an existing event-map file, see “The File Selection Dialog Box” on page 5-34.

For more information on event-map files, see “Understanding Event-Map Files” on page 5-10.

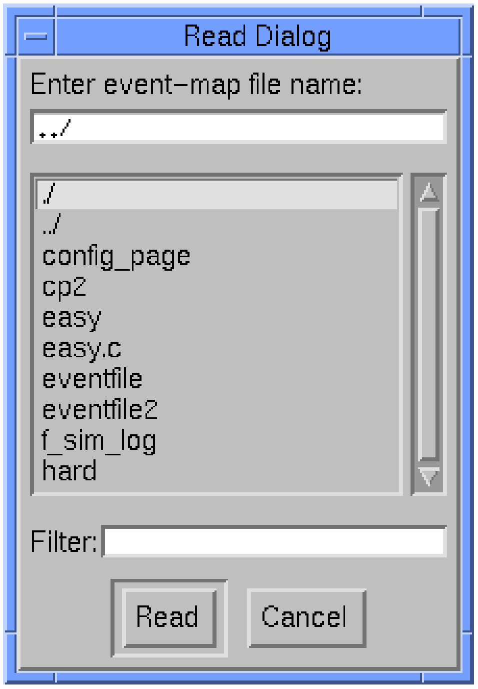


Figure 5-7. The Read Event-Map File Dialog Box

Exit

When you click on File Ì Exit on the Global Window, **`ntrace`**:

- Prompts you to save any unsaved changes, if appropriate
- Discards unsaved changes, if appropriate
- Deallocates memory it used to store trace events
- Exits

`ntrace` puts up a Warning Dialog Box if you try to exit from **`ntrace`** without saving your display page changes. If you want to save your changes, click on Cancel; as a result, **`ntrace`** removes the dialog box, and you can save your changes. If you do not want to save your changes, click on OK; this causes **`ntrace`** to exit.

Help Menu Item

When you click on the Help menu item on the Global Window, the pull-down menu shown in Figure 5-8 appears.

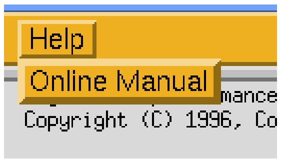


Figure 5-8. Global Window Help Menu

The Online Manual item opens the online version of the *NightTrace Manual* using the HyperHelp viewer that is shipped as part of the X Window System (**x11**) product.

The online *NightTrace Manual* can also be accessed using the **nhelp** utility shipped with the X Window System. The manual name is **ntrace**. For example, from the command line:

```
nhelp ntrace
```

opens the most recently installed version of the *NightTrace Manual* in the HyperHelp viewer.

The File Selection Dialog Box

The File Selection Dialog Box gives you three ways to find a file:

- Type in the exact file name
- Scroll through existing file names until you see the one you want
- Type in a filter (file name pattern) for **ntrace** to locate

Typing in the Exact File Name

If you know the exact file name, use the following steps to open the file.

1. Type a file name, possibly with leading directory name(s), into the File Name text field.
2. If you mistype the file name, see “Field Editing” on page 6-16 and correct the problem.
3. Press <Enter>.

This causes **ntrace** to remove the File Selection Dialog Box.

If you have read permission to this file and it is of the right type, **ntrace** opens the file. If it is a configuration file, **ntrace** puts up any display page(s) from this file. If it is an event-map file, **ntrace** adds those trace event tags and trace event argument formats to its internal list.

If opening the file was not successful, **ntrace** puts up a Warning Dialog Box. The warning message in the dialog box differs depending on the problem. When you have read the warning, click on OK. As a result, **ntrace** removes the dialog box.

If you cause **ntrace** to bring up this File Selection Dialog Box again, all fields contain the same values as when you left this dialog box, except the File Name text field never comes up with more than a directory name.

Scrolling Through Existing File Names

If you would recognize the file name if you saw it, use the following steps to find and open it.

1. Use the scroll bar to examine the alphabetical list of file and directory names displayed in the scroll region.
2. Try to find the file name you are seeking.
3. If you find the file:
   a. Click on the file name to select and highlight it.
   b. Click on Open.

As an alternative to these two steps, you could double click quickly on the file name.

If you have read permission to this file and it is of the right type, **ntrace** opens the file. If it is a configuration file, **ntrace** puts up all display page(s) from this file. If it is an event-map file, **ntrace** adds those trace event tags and trace event argument formats to its internal list.

If opening the file was not successful, **ntrace** puts up a Warning Dialog Box. The warning message in the dialog box differs depending on the problem. When you have read the warning, click on OK. As a result, **ntrace** removes the dialog box.

If you do not find the file in the list:

1. Click on the directory name under which it resides. This selects and highlights the directory name.
2. Click on Open.

As an alternative to these two steps, you could double click quickly on the directory name.

This causes **ntrace** to:

- Put the selected directory's name in the File Name text field
- Change to that directory (**cd**)
- Display the file and directory names under that directory

You can repeat the steps in this method until you find the file.

If you cause **ntrace** to bring up this File Selection Dialog Box again, all fields contain the same values as when you left this dialog box.

TIP:
Clicking on the `".."` directory causes the scrolled list to be filled with the contents of the parent directory.

Typing in a Filter (File Name Pattern)

If you know only some of the characters in the file name, use the steps below to find and open it.

1. Type a file name pattern, possibly with leading directory name(s) and appropriately-placed asterisk(s), into the `Filter` text field. Each asterisk (`*`) in this field represents zero or more characters at this position.
2. If you mistype the field name pattern, see "Field Editing" on page 6-16 and correct the problem.
3. Press `<Enter>`.

This causes **`ntrace`** to replace the contents of the scroll region with subdirectory names and file names that match your pattern. To locate your file in the scroll region and open it, see the "Scrolling Through Existing File Names" on page 5-35.

If you caused **`ntrace`** to bring up this `File Selection Dialog Box` again, all fields contain the same values as when you left this dialog box.

Exercise: Displaying Trace Events

The following exercise has you graphically display the trace events you logged in "Exercise: Logging Trace Events" on page 4-27.

Copy the **`/usr/lib/NightTrace/examples/entry_exit_page`** configuration file to your directory and call it **`page`**. (See "Understanding Page Configuration Files" on page 5-12 for more information about configuration files.)

```
$ cp /usr/lib/NightTrace/examples/entry_exit_page page
```

Copy the **`/usr/lib/NightTrace/examples/entry_exit_map`** event-map file to your directory and call it **`map`**. (See "Understanding Event-Map Files" on page 5-10 for more information about event-map files.)

```
$ cp /usr/lib/NightTrace/examples/entry_exit_map map
```

Invoke **`ntrace`** with the trace event file you created in the last exercise and the configuration file you just created.

```
$ ntrace log page
```

NightTrace presents a display page. Concentrate on the dotted grid area in the middle and the row of push buttons at the bottom. Keep clicking on the `Zoom Out` push button until the display quits changing. Click on the Ruler around 2 seconds. The display object with digital “waves” is a *StateGraph*. It graphically displays trace events and states. The two “floating” *DataBoxes* contain textual information about the current trace event and its first argument, respectively. Notice that the current trace event is identified by its cryptic trace event ID number. (See “StateGraph” on page 7-14 and “DataBox” on page 7-12 for more information about StateGraphs and DataBoxes.)

The next few steps get the same display page to show symbolic tags instead of numeric IDs for trace events.

Close the display page by clicking on File Ì Close. (See “Close” on page 7-19 for more information about this menu item.)

Read in the event-map file named **map** by clicking on File Ì Read Event-Map File …. (See “Read Event-Map File” on page 5-32 for more information about this menu item.)

Re-open the configuration file named **page** by clicking on File Ì Open Config File …. (See “Open Config File” on page 5-31 for more information about this menu item.) Click on the Refresh push button on the display page. Notice that the current trace event is now identified by its symbolic tag because that trace event has an entry in the **map** file.

This exercise continues in “Exercise: Using the Search Tool” on page 10-14.

For practice customizing the graphical user interface, read Appendix B and try “Exercise: Customizing Display Colors” on page B-5.

6
Viewing Trace Event Logs with ntrace

6
Viewing Trace Event Logs with ntrace

Overview

`ntrace`'s display page has two modes: Edit mode and View mode. The words "Edit" and "View" pertain to the operations you can perform on the graphical display, not the text fields or scroll bar. This chapter discusses *View mode*, the mode that displays trace events and states from your trace event file(s). **`ntrace`** displays this information:

- Graphically in configured display object(s) on the grid
- Statistically in fields of the interval control area
- Uniformly on all display page(s). (This means that changes on one page are reflected on all pages.)

`ntrace` uses the same *display page*(s) in both Edit and View modes. However, toggling between modes changes the interval scroll bar, fields in the interval control area, and the push buttons. In View mode, the message display area shows some statistics, as well as errors and warnings. The default mode for an existing display is View mode.

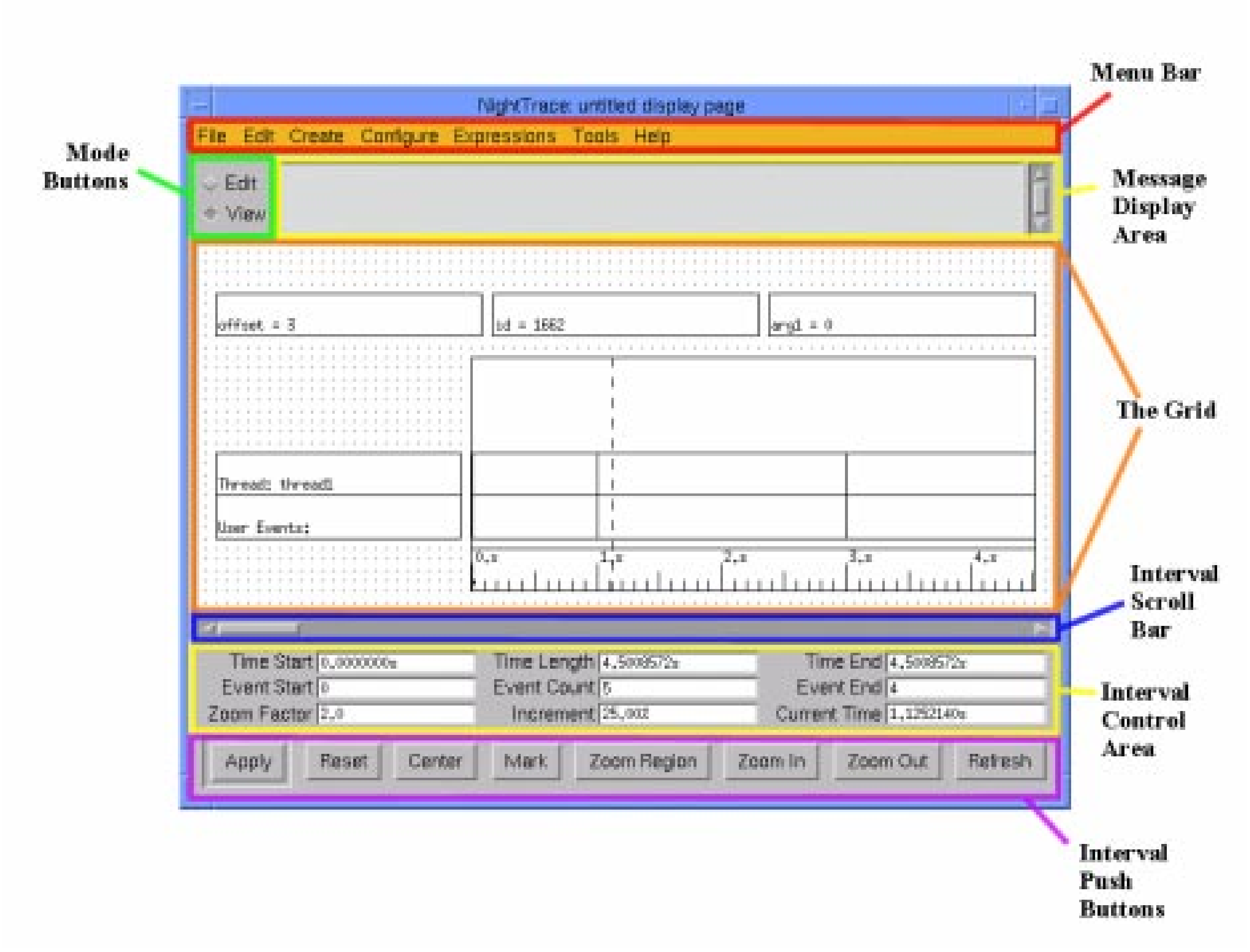
View mode lets you locate interesting parts of your trace session by:

- Shifting with the interval scroll bar
- Clicking on some of the interval push buttons
- Editing some field(s) in the interval control area
- Using the built-in Search tool (See Chapter 10 for more information.)

See Chapter 7 for more information on Edit mode, the components of the display page, and display objects.

This chapter assumes that you have already created or loaded a display page with configured display objects. This manual uses the following term conventions:

`<Enter>`	The key on your keyboard that issues a carriage return and line feed.
`<Backspace>`	The key on your keyboard that issues a `<Ctrl> <h>`. In **`ntrace`** this is also `<Delete>`.
interval	A time period in the trace session that has a specific starting and ending time. It is the "window" into the trace session that appears on the display page.



Time Start 0.0000000s Time Length 4.5008572s Time End 4.5008572s
Event Start 0 Event Count 5 Event End 4
Zoom Factor 2.0 Increment 25.002 Current Time 1.1252140s

Figure 6-1. A Display Page in View Mode

current time	The instance in time currently being displayed. It occurs within the interval. Searches begin at the current time.
current time line	The dashed vertical bar that represents the current time in a Column. This line moves to the location of your pointer when you click with mouse button 1 in a Column. The position of the current time line determines the values that appear on display pages.

This chapter covers the following topics:

- Mouse button operations in View mode
- Understanding the grid
- Deciding what to do next in View mode
- Using the interval scroll bar
- Using the interval push buttons
- Understanding the interval control area

- Field editing

Mouse Button Operations

Mouse button operations in View mode appear in Table 6-1 and in the *NightTrace Pocket Reference* card. Unfamiliar terminology is defined later in this chapter.

Table 6-1. View-Mode Mouse Button Operations

Button	Use Within a Column
Mouse button 1	Move the current time line to the place where the pointer rests, or put the text cursor where you clicked in the text field.
Hold down <Ctrl> and click mouse button 1	Move the mark and the current time line to the place where the pointer rests.
Hold down <Ctrl>, hold down mouse button 1, and drag horizontally	Move the mark to the beginning point of the drag region, and move the current time line to the ending point of the drag region. The drag region is highlighted as you drag the pointer.
Mouse button 2	Write a statistic in the message display area that tells about the trace event where the pointer rests in a StateGraph or EventGraph.
Hold down <Ctrl> and click mouse button 2	Write a statistic in the message display area that tells how far the pointer is from the mark. A positive number means the pointer is to the right of the mark. A negative number means the pointer is to the left of the mark.
Mouse button 3	Write a statistic in the message display area that tells about the data item where the pointer rests in a DataGraph.
Hold down <Ctrl> and click mouse button 3	Write a statistic in the message display area that tells how far the pointer is from the current time line. A positive number means the pointer is to the right of the current time line. A negative number means the pointer is to the left of the current time line.

The Grid

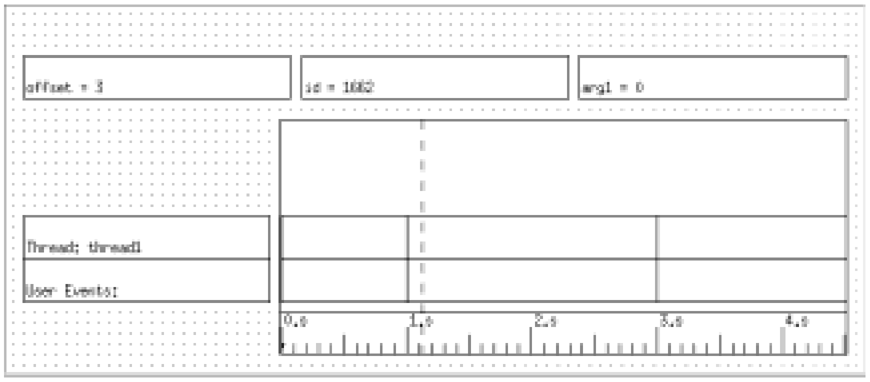


Figure 6-2. The Grid

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.

The `trace_open_thread()` routine and the **`ntraceud`** daemon write overhead trace events into your trace event file. The tags for these trace events are `NT_ASSOC_PID` and `NT_ASSOC_TID`. In View mode, you may see these trace events in display objects on the grid. **`ntrace`** 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 "The Interval Control Area" on page 6-11.

Some display objects on the grid contain vertical lines. Each vertical line in a StateGraph or EventGraph represents a user trace event, kernel trace event, or NightTrace overhead trace event. If you click on a trace event with mouse button 2, **`ntrace`** writes information about that trace event in the message display area. Each vertical line in a DataGraph represents a trace event argument. If you click on a data value with mouse button 3, **`ntrace`** writes information about the data value in the message display area. For information about StateGraphs, EventGraphs, and DataGraphs, see "StateGraph" on page 7-14, "EventGraph" on page 7-15, and "DataGraph" on page 7-16.

If your grid has a Column and you have not already positioned your interval somewhere else, **`ntrace`** 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 **`ntrace`** 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 Tools ➪ Search menu item to search for a specific trace event or condition. (See Chapter 10 for more information.)

Viewing Strategy

`ntrace` is a flexible tool. Depending on your personal preferences and how much you know about your trace events, there are several ways to locate intervals of interest. The following flowchart provides information to help you decide what to do next in View mode.

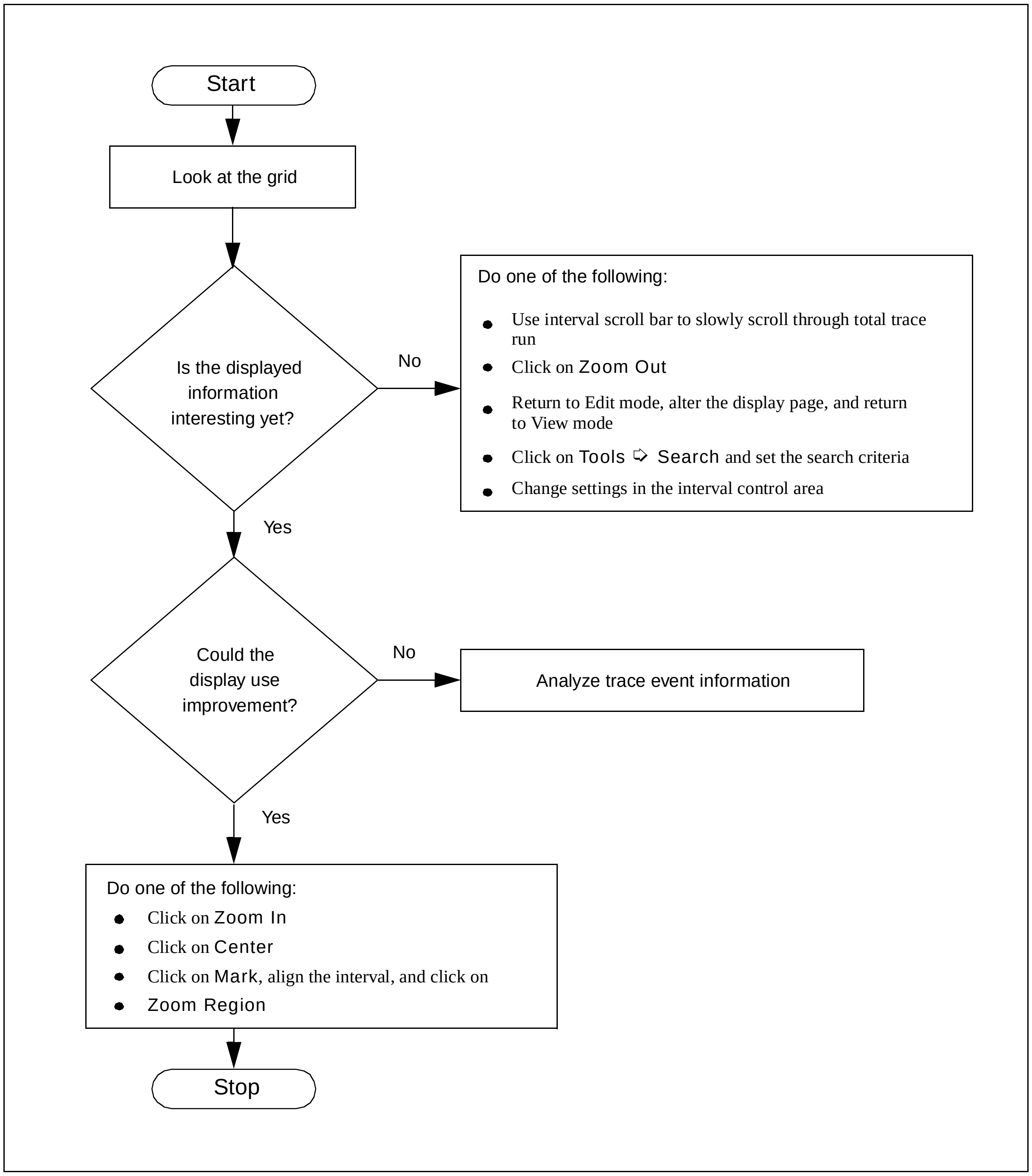

Start
Look at the grid
Is the displayed information interesting yet?
No
Do one of the following:
Use interval scroll bar to slowly scroll through total trace run
Click on Zoom Out
Return to Edit mode, alter the display page, and return to View mode
Click on Tools ➪ Search and set the search criteria
Change settings in the interval control area
Yes
Could the display use improvement?
No
Analyze trace event information
Yes
Do one of the following:
Click on Zoom In
Click on Center
Click on Mark, align the interval, and click on
Zoom Region
Stop

Figure 6-3. Deciding What to Do Next in View Mode

The Interval Scroll Bar

Although by its position it may look as if it scrolls the grid, the interval scroll bar scrolls the interval. 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. Changes in the display objects are most obvious when you have a Column that contains both a StateGraph and a Ruler. For more information on the interval control area, see “The Interval Control Area” on page 6-11. See Chapter 7 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 `Time Start` 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`, `Time Start`, `Time End`, `Event Start`, `Event End`, and `Increment`. For more information on these fields, see “The Interval Control Area” on page 6-11.

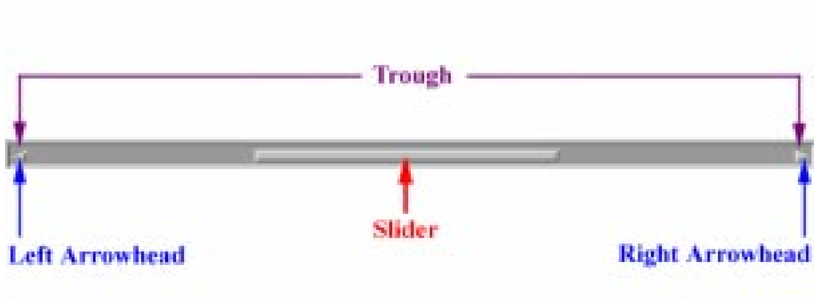


Figure 6-4. The Interval Scroll Bar

Manipulating the interval scroll bar in the following ways has the following results.

Table 6-2. 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

The Interval Push Buttons

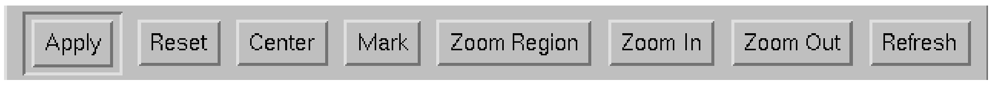


Figure 6-5. The Interval Push Buttons

The interval push buttons let you examine different intervals in your trace session. The eight push buttons appear just below the grid on the display page. In the following push button descriptions:

- Click on a push button by first pointing to it and then clicking with mouse button 1.
- Current Time, Time Start, Time End, Time Length, Event Start, Event End, and Event Count refer to fields in the interval control area.

Except for the Reset push button, each push button has an effect on:

- The fields in the interval control area
- The display objects on the grid
- The current time line on the grid

The effect of clicking on a particular push button appears next.

Apply (the default)

- Validates any field change(s) in the interval control area and takes appropriate action.
- Makes corresponding changes to other field(s).
- Possibly updates display objects on the grid.
- Possibly moves the current time line in a Column.
- Is equivalent to pressing <Enter>.

Reset

- Restores changed field(s) in the interval control area to the value(s) they had immediately after the last Apply or <Enter>. This works only if you have not already pressed <Enter> or clicked on the Apply push button.
- Is equivalent to pressing <Esc>.

Center

- Centers the interval around the current time line in a Column.
- Makes corresponding changes to Time Start, Time End, Event Start, and Event End.

Mark

- Sets a mark that points to a particular time. A mark is represented by a solid triangle on the Ruler. **ntrace** currently supports only one mark. By default this mark is at time 0.
- Puts a mark at the current time line of all Rulers.
- Is useful before clicking on Zoom Region.
- Can provide a statistic about the distance between your pointer and the mark.

Some control sequences pertain to the mark, the current time line, and your pointer.

- Simultaneously pressing <Ctrl> and clicking on mouse button 1 moves the mark and the current time line to the place where your pointer rests.
- Simultaneously holding down <Ctrl> and clicking on mouse button 2 causes **ntrace** to write a statistic in the message display area that tells how far your pointer is from the mark. A positive number means your

pointer is to the right of the mark. A negative number means your pointer is to the left of the mark.

- Simultaneously holding down <Ctrl> and clicking on mouse button 3 causes **ntrace** to write a statistic in the message display area that tells how far your pointer is from the current time line. A positive number means your pointer is to the right of the current time line. A negative number means your pointer is to the left of the current time line.
- Simultaneously holding down <Ctrl>, holding down mouse button 1, and dragging your pointer horizontally in a Column makes **ntrace** move the mark to the beginning point of the drag region and move the current time line to the ending point of the drag region. The region is highlighted as you drag the pointer.

Zoom Region

- Sets the interval to be the time between the mark and the current time line (inclusive).
- Sets Time Start to either the mark or the current time line, whichever is leftmost.
- Sets Time End to either the mark or the current time line, whichever is rightmost.
- Centers the current time line in a Column.
- Displays an error message in the message display area if the mark and the current time line are at the same place.

Zoom In

- Centers the interval around the current time line in a Column.
- Divides Time Length by the value of Zoom Factor; this provides a microscopic view of a smaller interval.
- Makes corresponding changes to Time Start, Time End, Event Start, Event Count, and Event End.

Zoom Out

- Centers the interval around the current time line in a Column.
- Multiplies Time Length by the value of Zoom Factor; this provides a macroscopic view of a larger interval.
- Makes corresponding changes to Time Start, Time End, Event Start, Event Count, and Event End.

Refresh

- Updates the grid to reflect the result of changes in configuration.
- Is implicit with any action that updates the grid.
- Should be used when you:

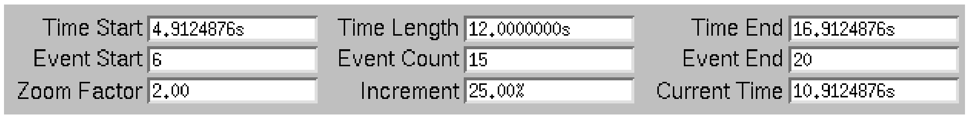
- Open a display page
- Switch to View mode from Edit mode
- Change a configuration parameter from View mode
- Resize the grid

• Differs from the X window manager's Refresh which redraws the windows without notifying **ntrace**.

The 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, **ntrace** 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



Time Start	4.9124876s	Time Length	12.0000000s	Time End	16.9124876s
Event Start	6	Event Count	15	Event End	20
Zoom Factor	2.00	Increment	25.00%	Current Time	10.9124876s

Figure 6-6. The Interval Control Area

Reading Fields

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 Time Start through Time End.

Time Start	Is the beginning time of the interval in seconds.
Time End	Is the ending time of the interval in seconds.
Time Length	Is the amount of time within this interval in seconds. It is the difference between Time End and Time Start.
Current Time	Is the present time within the interval in seconds.
Event Start	Is the ordinal number (offset), not the trace event ID, of the first trace event in this interval.

Event End	Is the ordinal number (offset), not the trace event ID, of the last trace event in this interval.
Event Count	Is the quantity of trace events present in this interval. It is the difference between Event End and Event Start plus one.
Zoom Factor	Is the number of times to magnify (or reduce) the interval each time you click on Zoom Out (or Zoom In). The default is 2.
Increment	Controls how much the current interval scrolls (and the slider moves) when you click on an arrowhead of the interval scroll bar or between an arrowhead and the slider on the interval scroll bar. This field may contain either a percentage or an absolute amount of time in seconds. The default is 25%.

Editing Single Fields

Changing the interval control area fields allows you to examine different intervals in your trace session. Usually you modify fields in the interval control area when you already know something about your trace events and their distribution.

When you press <Enter> or click on the Apply push button at the end of your editing, **ntrace** validates the data in each field you modified and takes appropriate action. If **ntrace** detects an invalid value, it restores the affected field to its previous value. For more information on the Apply push button, see “The Interval Push Buttons” on page 6-8.

ntrace displays all times in the interval control area in seconds with the “s” suffix. You can enter times into time-related fields in the following ways:

- Numeric time. **ntrace** assumes that the time unit is seconds.
- Numeric time in seconds with a “s” suffix.
- Numeric time in microseconds with a “u” suffix.

The following text explains what constitutes a valid field change and describes the effects of changing a single field. For general information on field editing, see “Field Editing” on page 6-16.

Time Start	A valid change keeps Time Start 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 Time End, change appropriately. If you set Time Start to the word `start`, **ntrace** resets Time Start to the start time (0 microseconds) of the trace session.
Time End	A valid change keeps Time End 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 Time Start, change appropriately.

If you change Time End so it is smaller than Time Length, **ntrace** sets Time End to Time Length. If you set Time End to the word end or an arbitrarily large number, **ntrace** resets Time End to the last time recorded in the trace event file(s) and changes other fields appropriately.

Time Length

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. Time End and other fields change appropriately.

If you set Time Length to the word all or an arbitrarily large number, **ntrace** resets Time Length to the last time recorded in the trace event file(s) and changes other fields appropriately.

Current Time

The *current time* is the specified time.

If the new current time is inside the current interval, the current time line moves appropriately in any Columns 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.

Event Start

A valid change keeps Event Start 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 Event Start to the word start, **ntrace** resets Event Start to 0 and Time Start to 0 microseconds.

Event End

A valid change keeps Event End 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 Event End to the word end, or an arbitrarily large number, **ntrace** resets Event End to the total number of trace events in your trace event file(s).

Event Count

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, **ntrace** resets Event Count to the total number of trace events in your trace event file(s) and changes other fields appropriately.

Zoom Factor

A valid change keeps Zoom Factor greater than or equal to 1. If you set Zoom Factor to the word default or a space, **ntrace** resets Zoom Factor to the default value, 2.

Increment	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, **ntrace** resets Increment to the default value, 25%. 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. However, if Increment is equal to 100%, you see a completely new interval. For more information on the interval scroll bar, see "The Interval Scroll Bar" on page 6-7.

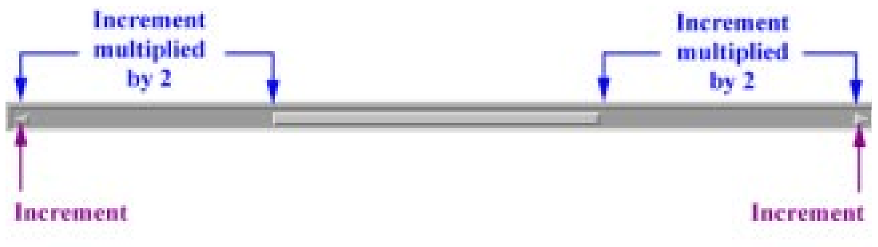


Figure 6-7. Amount of Scrolling Due to Increment Value

Editing Multiple Fields

Sometimes it makes sense to change multiple fields for a single effect; for example, you may wish to change both the Time Start and Time End fields or you may wish to change both the Time Start and Event Count fields. In these cases, apply your changes only once, after you have edited each field of interest.

Changing some combinations of fields is not meaningful; for example, you may try to change both Time Length and Event Count. When **ntrace** detects a meaningless combination of changes, it displays an error message in the message display area and restores the affected fields to their previous values. When **ntrace** detects an invalid value, it restores the affected field to its previous value.

Some general rules apply to multiple field editing.

- You must not simultaneously apply changes to more than two trace event fields.
- You must not simultaneously apply changes to more than two time fields; for these purposes Current Time is not considered to be a time field.
- You can change Current Time with any other valid field changes as long as Current Time falls within the new interval.
- You can change Zoom Factor with any other valid field changes.

- You can change Increment with any other valid field changes.
- Simultaneously modifying one time field and clearing another time field makes **ntrace** use the static and modified fields to determine the values of the cleared time field and the other fields.
- Simultaneously modifying one trace event field and clearing another trace event field makes **ntrace** use the static and modified fields to determine the values of the cleared trace event field and the other fields.

The following table shows all the valid multiple field changes except those that involve Current Time, Zoom Factor, or Increment. For information on editing specific fields of the interval control area, see “The Interval Control Area” on page 6-11. For general information on field editing, see “Field Editing” on page 6-16.

Table 6-3. Valid Multiple Field Changes

Fields	Result
Time Start Time End	The new interval starts at Time Start and ends at Time End.
Time Start Time Length	The new interval starts at Time Start and has a length of the specified Time Length.
Time Length Time End	The new interval ends at Time End and has a length of the specified Time Length.
Event Start Event End	The new interval starts at ordinal trace event number (offset) Event Start and ends at ordinal trace event number (offset) Event End.
Event Start Event Count	The new interval starts at ordinal trace event number (offset) Event Start and includes the specified quantity of trace events.
Event Count Event End	The new interval ends at ordinal trace event number (offset) Event End and includes the specified quantity of trace events.
Time Start Event Count	The new interval starts at Time Start and includes the specified quantity of trace events unless the Time Length forces Time Start to change.
Time End Event Count	The new interval ends at Time End and includes the specified quantity of trace events unless the Time Length forces Time End to change.
Event Start Time Length	The new interval starts at ordinal trace event number (offset) Event Start and has a length of the specified Time Length unless the Time Length forces Event Start to change.
Event End Time Length	The new interval ends at ordinal trace event number (offset) Event End and has a length of the specified Time Length unless the Time Length forces Event End to change.

Field Editing

You make changes to fields by following these steps:

1. Do one of the following:
   - Click with a mouse button on the field you want to edit. Clicking with mouse button 1 leaves a blinking vertical bar called the *text cursor* where you clicked in the field. Clicking with the other mouse buttons leaves the text cursor at the end of the field.
   - Drag with mouse button 1 on the field you want to edit.
   - If there already is a text cursor in a field, you can press <Tab> to move to the next field or <Shift> <Tab> to move to the previous field.
2. Use the built-in field editor to change values. Editing procedures follow.
3. Either press <Enter> or click on the Apply push button. This is called *applying your changes*.

Editing Text Fields

You can make the following types of editing changes in a text field:

- Insert text
- Delete text
- Replace text
- Undo a text change

Table 6-4. Making Editing Changes

Goal	Steps to Attain Goal
Insert character(s)	1. Position the text cursor where you want to insert character(s). 2. Type in the additional character(s).
Delete one character to the right	1. Position the text cursor to the left of the character to be deleted. 2. Simultaneously press <Ctrl> <d>.
Delete one character to the left	1. Position the text cursor to the right of the character to be deleted. 2. Either press <Backspace>, <Delete>, or simultaneously press <Ctrl> <h>.

Table 6-4. Making Editing Changes

Goal	Steps to Attain Goal
Delete adjacent character(s)	1. Point to the first character to be deleted. 2. Drag the pointer across any other characters to be deleted, release the mouse button, and keep the pointer in the field. This highlights the characters you dragged the pointer across. 3. Either press <Backspace>, <Delete>, or simultaneously press <Ctrl> <h>.
Replace adjacent character(s)	1. Point to the first character to be replaced. 2. Drag the pointer across any other characters to be replaced, release the mouse button, and keep the pointer in the field. This highlights the characters you dragged the pointer across. 3. Type in the new character(s).
Replace all character(s)	1. Position the text cursor anywhere in the field you want to modify. 2. Simultaneously press <Ctrl> <u>. This highlights all characters in the field. 3. Type in the new character(s).
Restore the default value	1. Replace all character(s) in the field with either a single space character or the word default. Note: Some fields do not have default values. 2. Press <Enter> or click on Apply.
Undo editing change(s) since the last <Enter> or Apply	1. Position in the window you want to modify. 2. Press <Esc> (or click on Reset if this is available).

Sometimes it is desirable to change multiple fields before applying the changes. In these cases, apply your changes only once, after you have edited each field of interest.

When you press <Enter> or click on Apply at the end of your editing, **ntrace** validates the data in each field you modified. **ntrace** rarely issues error messages about editing errors it detects. Usually it takes a default action. Some of the default actions include:

- If you enter an invalid value, for example alphabetic characters in a numeric field, **ntrace** ignores the changes and restores the previous values.
- Usually, if you enter a number that exceeds the maximum value, **ntrace** replaces it with the maximum value.
- If a range's starting value exceeds its ending value, **ntrace** swaps them.

Positioning Within Text Fields

You can either position the text cursor to a particular place within a field by either clicking or typing in key sequences. The following key sequences move the text cursor only if you are already positioned in a text field.

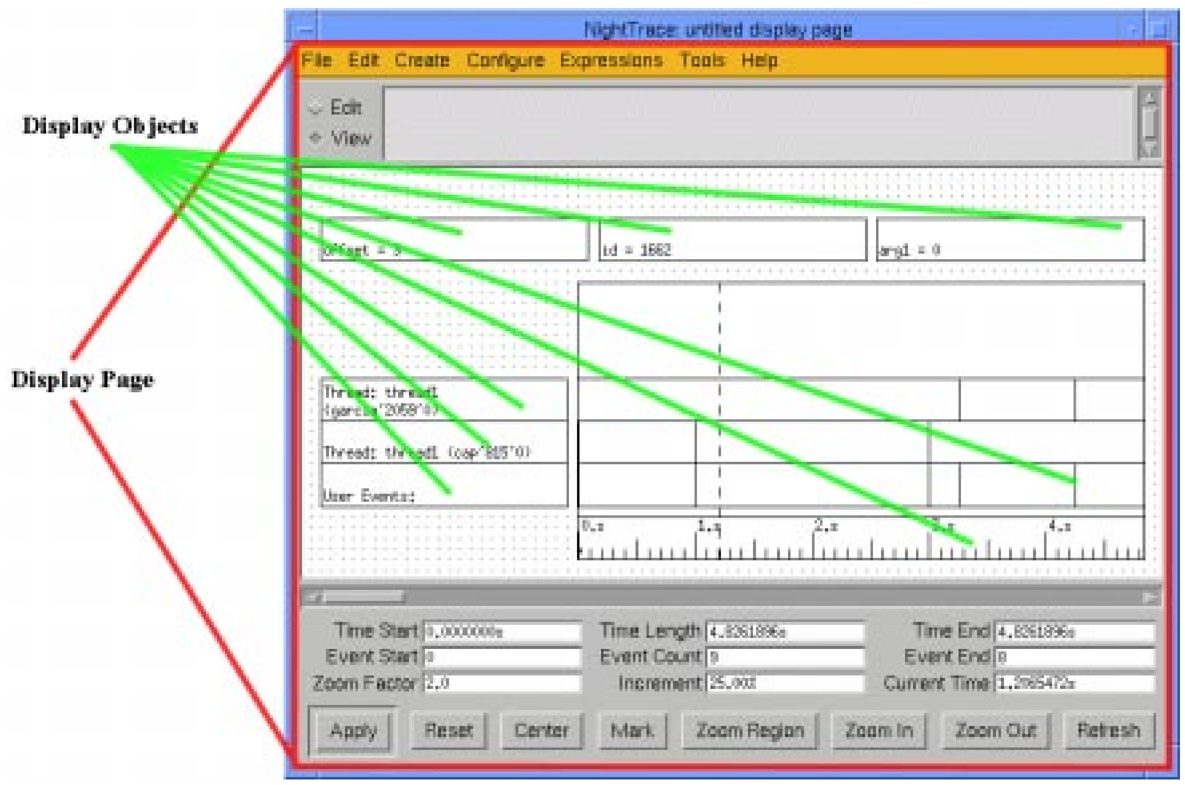
Table 6-5. Positioning Within a Text Field

Goal	Steps to Attain Goal
Move text cursor left one character	Press <LeftArrow> or simultaneously press <Ctrl> <b>. This action may cause scrolling.
Move text cursor right one character	Press <RightArrow> or simultaneously press <Ctrl> <f>. This action may cause scrolling.
Move text cursor to next field	Press <Tab>.
Move text cursor to previous field	Press <Shift> <Tab>.

7
Creating Display Objects

7
Creating Display Objects

Overview


Display Objects
Display Page
NightTrace: untitled display page
File Edit Create Configure Expressions Tools Help
Edit
View
Apply Reset Center Mark Zoom Region Zoom In Zoom Out Refresh

Offset = 3
id = 1662
arg1 = 0
Thread: thread1 (garcia'2059'0)
Thread: thread1 (cap'815'0)
User Events:

Time Start 0.0000000s
Time Length 4.8261896s
Time End 4.8261896s
Event Start 0
Event Count 9
Event End 8
Zoom Factor 2.0
Increment 25.000
Current Time 1.2065472s

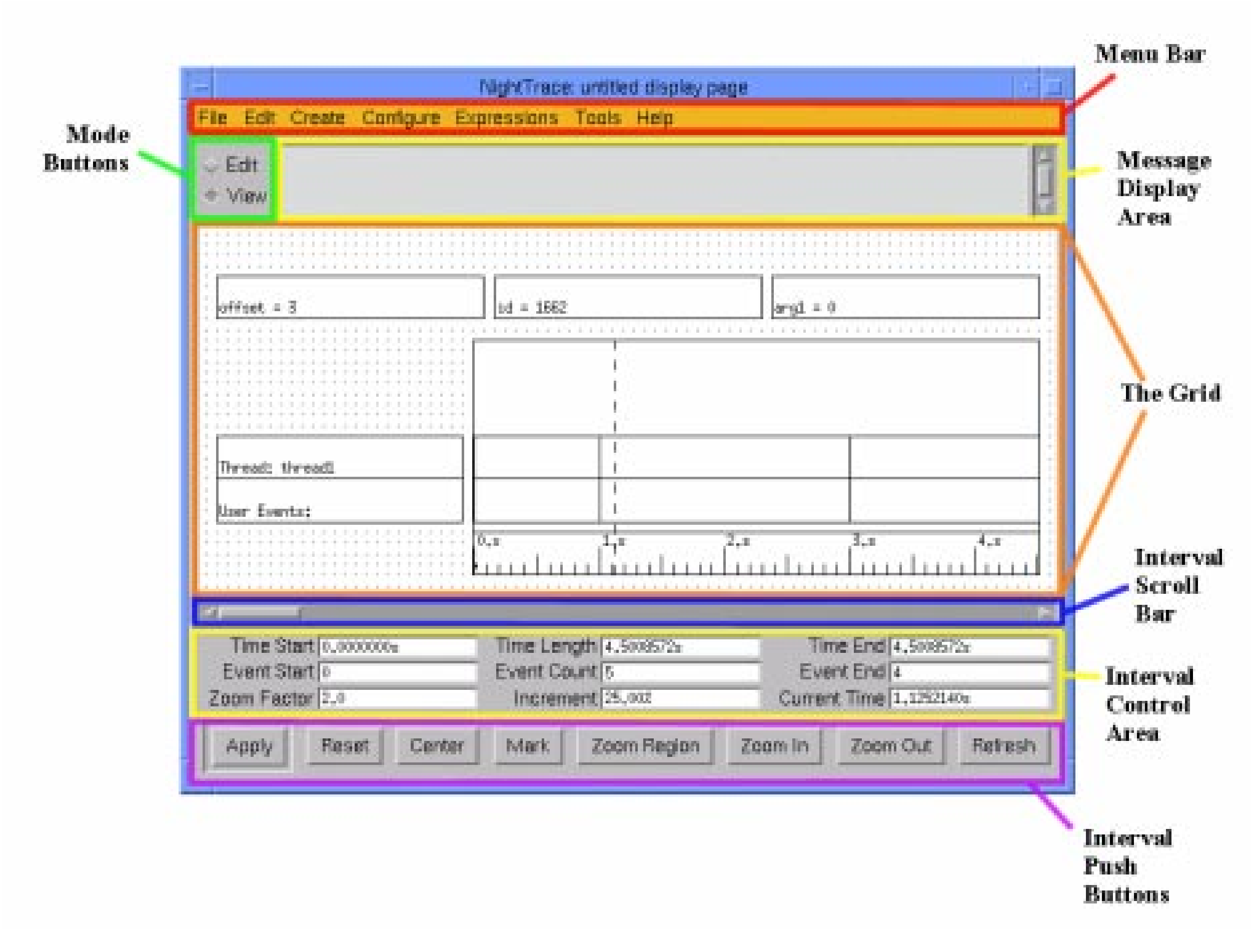
Figure 7-1. Display Page with Display Objects

Figure 7-1 shows what a display page may look like when you invoke **`ntrace`** and specify the default display page. The default display page contains display objects. (See "Default Page" on page 5-29.) *Display objects* filter, process, and display the information in the trace event file. These display objects are created with the display page and then viewed on the display page. You may want to create your own set of display objects to view your trace event file. To do this, follow the steps below.

1. Read "The Display Page" on page 7-2 to learn about the various parts of a display page.
2. Read "The Display Page" on page 7-2, which describes the different modes a display page can be in: Edit and View.

3. Put the display in Edit mode.
4. Read "Display Objects" on page 7-8, which explains what a display object is and what the different types of display objects you can put on your display page are.
5. Read "Operations on Display Objects" on page 7-4, which explains how to perform various operations (creating, selecting, moving and resizing) on display objects.
6. Create the various display objects you want and place them on the display page. Move or resize any display objects necessary to improve the layout of the page.

The Display Page



offset = 3
id = 1662
arg1 = 0
Thread: thread1
User Events:

Time Start 0,0000000s
Time Length 4,5008572s
Time End 4,5008572s
Event Start 0
Event Count 5
Event End 4
Zoom Factor 2,0
Increment 25,002
Current Time 1,1252140s

Figure 7-2. Elements of a Display Page

A *display page* lets you view the trace event data in the trace event file. Figure 7-2 shows an example of a display page and points out the portions of the display page. Following is a brief description of the portions of a display page:

Menu bar	Contains menu items. When you click on a menu item in the menu bar, a pull-down menu appears with a list of related menu entries. You can then initiate an operation on the menu.
Mode buttons	Are radio buttons that control whether the display page is in Edit or View mode and allow you to switch between modes by clicking on them.
Message display area	Displays error and status messages. It has a scroll bar so you can view previous or current messages.
Grid	Contains display objects. Figure 7-2 shows a grid before any display objects have been created.
Interval control area	Contains information on the current interval being displayed and the controls to manipulate the display.

Display Page Modes

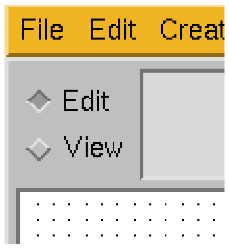


Figure 7-3. Edit and View Mode Buttons

Display pages can be operated in one of two modes: Edit mode or View mode. *Edit mode* lets you make changes to the display objects. *View mode* lets you view the execution of your application via the trace event file. The buttons for Edit and View mode are in the upper left-hand corner of the display page. If the display is in Edit mode, the button beside the word “Edit” is depressed. Otherwise, the View button is depressed and the display will be in View mode. To change modes, click with any mouse button on the button beside the desired mode.

Edit Mode

When the display page is in Edit mode, you can perform any of the operations on the menu bar except Tools, which is disabled (dimmed). The interval scroll bar, push buttons, and fields in the interval control area are disabled too.

View Mode

Once you have created a set of display objects and configured them, you can view the trace event information in the trace event file.

To view the data in the trace event file, the display page must be in View mode. However, if the display page is in View mode, you will not be able to create, edit, or configure display objects. See Chapter 6 for information on running (viewing) a display page.

Operations on Display Objects

This section describes some operations you can perform on display objects. The four operations discussed are:

- Creating new display objects and placing them on the grid
- Selecting display objects
- Moving display objects around the grid
- Resizing display objects

Each of these operations involves using the mouse buttons and the grid. Figure 7-4, Table 7-1, and the *NightTrace Pocket Reference* card show which mouse buttons correspond to which operations. These operations are referred to as *grid operations*. You can perform other operations on display objects using the Edit and Configure menus. Edit operations are discussed later in this chapter. See Chapter 8 for more information on configure operations.

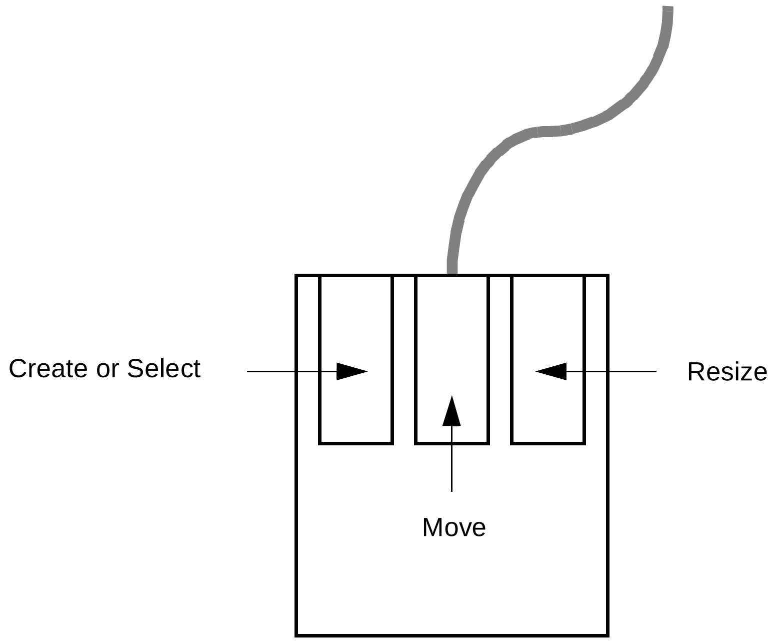


Figure 7-4. Button Functions on a Mouse

Table 7-1. Edit-Mode Mouse Button Operations

Button	Use Within the Grid
Mouse button 1	Create new objects, single select by clicking, or multiple select by dragging
<Ctrl> mouse button 1	Select the Column display object
<Shift> mouse button 1	Multiple select or toggle selection
Mouse button 2	Move display objects
<Ctrl> mouse button 2	Move the Column display object
Mouse button 3	Resize display objects
<Ctrl> mouse button 3	Resize the Column display object

Creating Display Objects

Before you can do any of the other operations, you must first create a display object. When you create a display object, you choose its place on the grid and its size.

Creating display objects involves three steps: selecting (*loading*) 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.

Some display objects go only inside of other display objects. StateGraphs, EventGraphs, DataGraphs and Rulers go only inside a Column.

To create a display object and place it on the grid, do the following:

1. Place the pointer on the `Create` entry on the menu bar and click mouse button 1.
2. Select the type of display object you want to create. Note that the pointer is now a crosshair. The display object is now “loaded.”
3. 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.
4. 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).
5. 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 DataBox on top of an existing GridLabel). 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.

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 described in “Select All” on page 7-18.

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 7-6.

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.

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, except certain display objects must be placed inside a Column. If you try to move a display object on top of another display object, **`ntrace`** 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 7-6 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.

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 StateGraphs, EventGraphs, DataGraphs and Rulers that it contains.

Display objects must not overlap, with the exception that certain display objects need to be placed inside a Column. If you try to resize a display object on top of another display object, **ntrace** displays an error message in the message display area and aborts the resize.

Display Objects

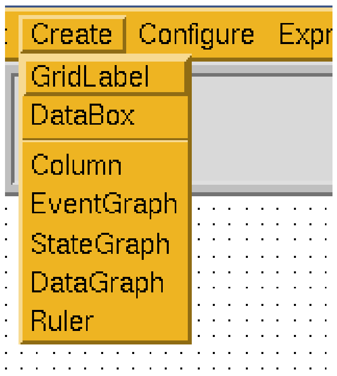


Figure 7-5. Create Display Objects Menu

Display objects, which are created via the `Create` menu shown in Figure 7-5, can be thought of as combination filters and formatters for the data stored in the trace event file. Every time a display object is updated, it filters through the data in the trace event file. 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, and optional arguments. NightTrace also keeps track of the ordinal number (offset) of a trace event. You can use **ntrace** functions to express any of these values. For more information about functions, see “Functions” on page 9-9.

Although the trace event file contains trace events, it also implicitly contains states. The concepts of trace events and states are key to understanding display objects.

trace event Corresponds to the point in the execution of your application when a `trace_event()` call was executed. All the data logged at that time (trace event ID, arguments, etc.) is considered a trace event.

state A state is bounded by two trace events, a start *event* and an end *event*. An instance of a state is the period of time between the start event and end event, including the start and end events themselves. Additional conditions may be specified in a state definition to further constrain the state. Instances of 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 process the information that you want displayed. This allows you to watch only the data of interest. Of course, all of this is dependent on the application having the necessary `trace_event()` calls enabled and inserted in the appropriate places.

All display objects are rectangular, but you specify the dimensions of the rectangle. Other properties of display objects you should be aware of are:

- 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 what point in the execution of the application you are looking at. *Static* means the contents do not change. All display objects except the GridLabel and the Ruler are dynamic.
- Display objects can be configurable or non-configurable. *Configurable* means you define the parameters that determine the content of the display object. *Non-configurable* means the display object has no parameters. All display objects except the Column are configurable.
- 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.

The basic types of display objects you can create are listed below and discussed in the following sections.

GridLabel	Static textual display object that contains a user-specified string of text and is used to label other display objects for clarity.
DataBox	Dynamic display object that displays textual information, such as the trace event tag or the time the trace event occurred. Its main use is to display data that does not lend itself to graphical representation.
Column	Dynamic display object that does not display data itself but holds the scrollable graphical display objects: StateGraphs, EventGraphs, DataGraphs, and Rulers. Its purpose is to group together related graphical display objects. It is the only non-configurable display object.
StateGraph	Dynamic, scrollable, graphical display object that displays a state as a bar and other trace events as a vertical line. It indicates the states' and trace events' relative positions in time 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.
EventGraph	Dynamic, scrollable, graphical display object that displays a trace event as a vertical line and indicates its relative position in time since the trace started. Use this display object if you want to know when particular trace events occur.
DataGraph	Dynamic, scrollable, graphical display object that displays a data as a vertical line or bar and indicates its relative position in time since the trace started. The height of the line or bar can be made proportional to the value of a trace event argument or other data. Use this display

	object to display relative values of arguments in the trace event record.
Ruler	Static, scrollable, graphical display object resembling a Ruler that displays the time. Rulers are used with StateGraphs, EventGraphs, and DataGraphs to show what time a trace event occurred.

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; display object type, size, configuration, and position have no bearing.

Display objects just created in Edit mode contain little useful information. The illustrations of display objects in this chapter show the display objects in View mode.

Figure 7-6 contains a flowchart to help you decide what display objects suit your needs. To use the flowchart, decide what type of information you want to display. Then start at the upper left-hand corner of the chart in the box labeled “Start.”

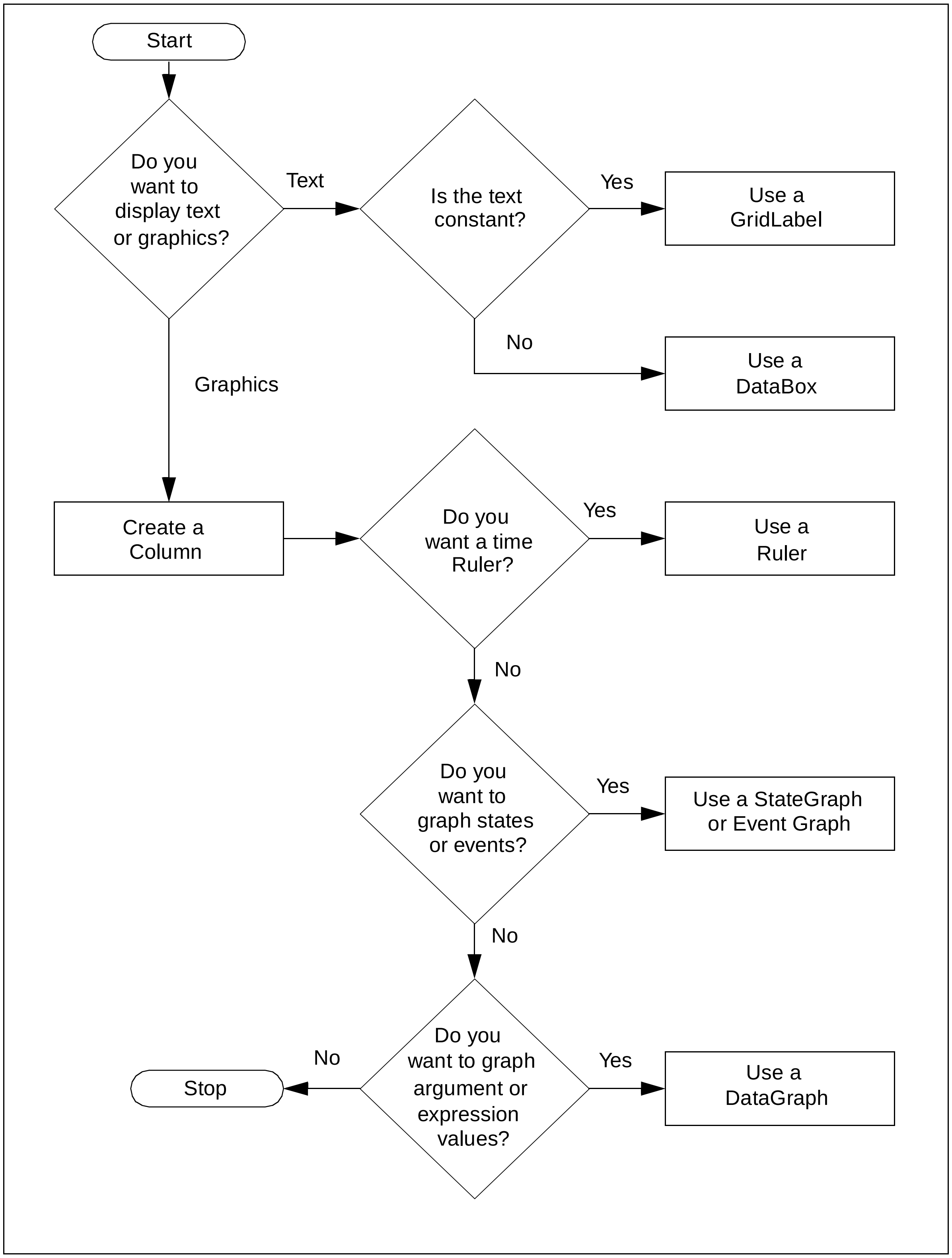

Start
Do you want to display text or graphics?
Text
Is the text constant?
Yes
Use a GridLabel
No
Use a DataBox
Graphics
Create a Column
Do you want a time Ruler?
Yes
Use a Ruler
No
Do you want to graph states or events?
Yes
Use a StateGraph or Event Graph
No
Do you want to graph argument or expression values?
Yes
Use a DataGraph
No
Stop

Figure 7-6. Display Object Use Flowchart

GridLabel

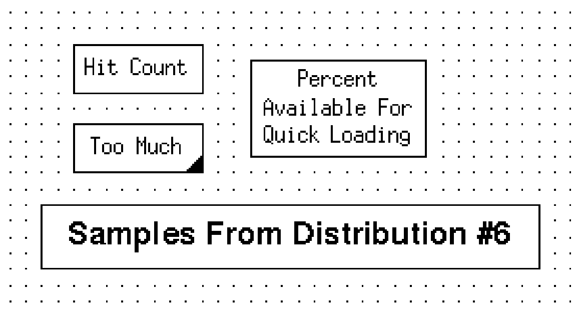

Hit Count
Too Much
Percent Available For Quick Loading
Samples From Distribution #6

Figure 7-7. GridLabel Examples

Clicking on `Create` ➪ `GridLabel` lets you draw or create a GridLabel display object on the grid. A *GridLabel* 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. GridLabels can appear anywhere on the grid, but they cannot go inside a Column. You can put several GridLabels on a grid.

If the text is too long to fit into the GridLabel, the lower right corner of the box is filled in. If this occurs, you should resize the GridLabel. This is described in “Resizing Display Objects” on page 7-7. A newly created label contains the word `label`. See “GridLabel” on page 8-12 for more information.

GridLabels are static display objects. That is, a GridLabel does not change its appearance or contents depending on the trace event data.

In addition to specifying the text inside of the GridLabel, 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 Chapter 8 for more information.

DataBox

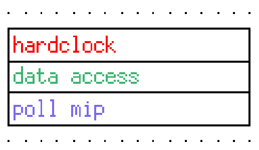


Figure 7-8. DataBox Examples

Clicking on `Create` ➪ `DataBox` lets you draw or create a DataBox display object on the grid. A *DataBox* 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.

DataBoxes are useful when you want to display data that does not lend itself to graphical representation, as shown in Figure 7-8. This figure shows three databoxes: the top DataBox contains the interrupt name, the middle contains the exception name and the bottom contains the syscall name. If the value is too large to fit into the DataBox (e.g., a long trace event tag), the lower right corner of the box is filled in. If this occurs, you should resize the DataBox. This is described in "Resizing Display Objects" on page 7-7. By default, numeric data is displayed in decimal integer. (For information about overriding this default, see "Understanding Event-Map Files" on page 5-10, "format()" on page 9-80, and "get_format()" on page 9-79.) A newly created DataBox contains a `0`. See "DataBox" on page 8-13 for more information.

DataBoxes can appear anywhere on the grid except within a Column. You can put several DataBoxes on a grid.

Some examples of data that you can configure a DataBox to show are:

- The tag of the last trace event before the current time (See Table 8-3.)
- The NightTrace thread name of the last trace event before the current time (See Table 8-3.)
- A particular argument logged with the last trace event before the current time (See "arg()" on page 9-20.)
- The total amount of time the application was in a particular state before the current time (See "state_dur()" on page 9-57 and "sum()" on page 9-72.)
- The number of times a particular trace event has occurred before the current time (See "event_matches()" on page 9-33.)
- A string of characters generated by a format expression (See "format()" on page 9-80.)

Column

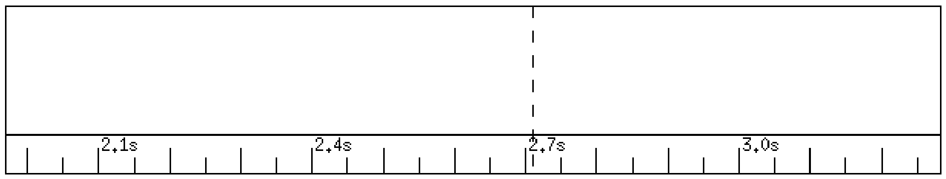


Figure 7-9. Column Example

Clicking on `Create` ➪ `Column` lets you draw or create a Column display object on the grid. When a *Column* is first created, it is an empty rectangle that does not display data of its own. A Column holds StateGraphs, EventGraphs, DataGraphs and Rulers. It provides a convenient way of associating these graphical display objects. Figure 7-9 shows a Column after a Ruler has been added.

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 StateGraphs, EventGraphs, DataGraphs 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.

TIP:

On a monochrome display, make sure that you can differentiate among display objects within a Column. The easiest way to do this is to leave at least one grid dot between display objects in a Column and to make the background color of the Column black. For more information on setting a Column's background color, see "Default X-Resource Settings for ntrace" on page B-2.

StateGraph

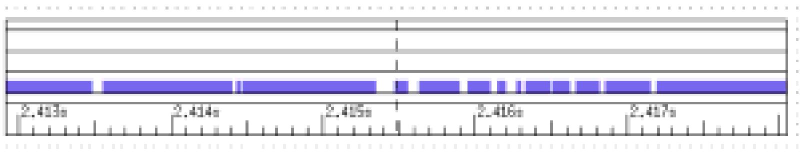


Figure 7-10. StateGraph Example

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. A *StateGraph* 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. 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. You can create a StateGraph by clicking on Create ➪ StateGraph and drawing on the grid.

StateGraphs must be placed in a Column. A StateGraph and a Ruler are shown in Figure 7-10.

A StateGraph can display trace events in a manner identical to an EventGraph. 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 StateGraphs 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

See “StateGraph” on page 8-14 for more information.

In View mode, 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.

EventGraph

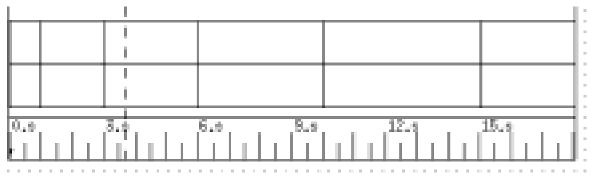


Figure 7-11. EventGraph Example

Clicking on Create ➪ EventGraph lets you draw or create an EventGraph display object on the grid. An *EventGraph* represents trace events as a thin vertical line. EventGraphs must be placed in a Column. Figure 7-11 shows an EventGraph with a Ruler below it.

Some examples of information that an EventGraph 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

See “EventGraph” on page 8-16 for more information.

In View mode, 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.

DataGraph

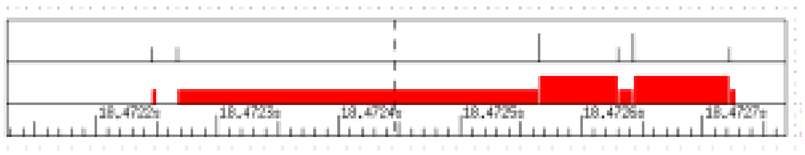


Figure 7-12. DataGraph Examples

Clicking on Create ➪ DataGraph lets you draw or create a DataGraph display object on the grid. DataGraphs must be placed in a Column. They represent data as either vertical lines or bars of varying height. In Figure 7-12 the same set of data is used to draw the two basic types of DataGraph. The top DataGraph is a line DataGraph, which shows the data as vertical lines of varying height. The bottom DataGraph is a bar DataGraph, which consists of bars of varying height. The height of the line or bar is proportional to data from the trace event file. This display object is usually used to display values of arguments in the trace event record.

Some examples of ways that a DataGraph 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 DataGraphs, make sure that they are high enough for you to distinguish differences in data values. See “DataGraph” on page 8-17 for more information.

TIP:
The higher you make the DataGraph, the easier it is to differentiate similar data points.

In View mode, 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.

Ruler

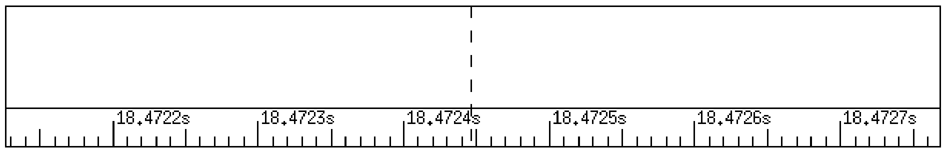


Figure 7-13. Ruler Example

The interval control area, which is described in “The Interval Control Area” on page 6-11, has three numeric fields that list the beginning, end, and current time for the time interval displayed in the Column. A *Ruler* display object, however, displays this information in a graphical format on the grid. Like their physical counterparts, Ruler display objects have major and minor hash marks to mark divisions, but the units are of time, not distance. They represent the amount of time since the first trace event was logged. Usually the first trace event is logged by the `trace_open_thread()` call. You can create a Ruler by clicking on `Create` ⇨ `Ruler` and drawing on the grid.

In addition to hash marks and numbers, Rulers can also have lost-data indicators and a mark. The lost-data indicator is a reverse-video “`L`” and indicates the location in time where NightTrace lost some data. For more information on trace event loss, see “Preventing Trace Events Loss” on page A-1. Marks are explained in “The Interval Push Buttons” on page 6-8.

Rulers are static display objects. That is, they do not change their appearance or contents depending on the trace event data. They do change their appearance, however, to reflect the current interval being displayed.

A Ruler should be at least three grid dots high. In addition to determining the size of the Ruler, you also specify other aspects of the Ruler. See “Ruler” on page 8-19 for more information.

Editing Operations

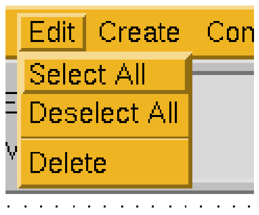


Figure 7-14. Edit Menu

Editing operations are enabled only when the display page is in Edit mode, which is selected by clicking on the radio button labeled "Edit" in the upper left-hand corner of the display page.

Select All

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).

Deselect All

Deselect All deselects every selected display object on the grid.

Delete

Delete deletes the selected display object(s).

File Operations

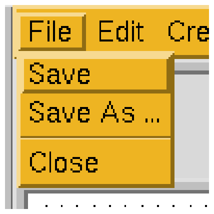


Figure 7-15. File Menu

The file operations are accessed through the File operations menu shown in Figure 7-15.

Save

Save saves the current display page (including all local macros, qualified events, and qualified states) to the configuration file you opened. Thus, any changes you have made since the last Save operation will be saved. You can continue editing or viewing the display after this operation. The Save operation is disabled (dimmed) if this is a new display page, or you have not made any changes since the last time the display page was saved. Instead, use Save As ….

Save As ...

Save As … saves the current display page to a file other than the one you opened. You can continue editing or viewing the display after this operation.

Save As … uses a File Selection Dialog Box to prompt you for a file name. See "The File Selection Dialog Box" on page 5-34 for more information.

Close

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 and you do a Close, a Warning Dialog Box appears, reminding you that you may want to save you changes.

8
Configuring Display Objects

8
Configuring Display Objects

Overview

Customizing a display object so that it displays only the information you want it to – in the way that you want it to – is called *configuring*. Configuring is done with the Configure ➪ Content menu item shown in Figure 8-1.

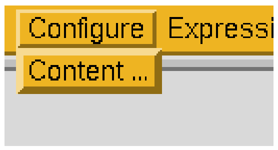


Figure 8-1. Configure Command Menu

Sections on configuring display objects discuss the following topics:

- Configuration parameters that are common to many display objects
- Operations you can perform on the configuration data
- Configuration parameters that are specific to each type of display object

NOTE

Columns are the only display objects that are not configurable.

Common Configuration Parameters

Different types of configuration parameters exist. Some parameters are concerned with how the information appears in the display object. These parameters are Foreground Color, Background Color, Font, Text Justify, Text Gravity, Fill Style, Event Color, Lost Event Color, Mark Color, Maximum, and Minimum. For each configuration parameter that pertains to color, there is an equivalent X resource. See Appendix B for more information.

Other parameters are concerned with determining the content of the information in the display objects. The parameter that does this is Then-Expression.

The last type of parameter is concerned with constraining the information that appears in the display object. These parameters act as filters, allowing only data that meets certain criteria to be displayed. These parameters are Event List, If-Expression, CPU List, PID List, TID List, Start-Events, End-Events, Start-Expression, and End-Expression.

The configuration parameters are changed with the same editing methods used in the interval control area. See “Field Editing” on page 6-16 for more information. Note that you can type `default` or just a space in a field to get the default value.

Many of the display objects share common configuration parameters. These common configuration parameters are summarized in Table 8-1 and discussed in the following sections. For more information about configuration parameters, refer to the sections on configuring the object you are interested in.

Table 8-1. Common Configuration Parameters

Parameter Name	Possible Values	Meaning
Display Object Name	Any alphanumeric string beginning with a letter. Underscores are also allowed. Spaces are not allowed.	The name of the display object.
Event List	Any meaningful combination of the following: • `ALL` • `ALLUSER` • `ALLKERNEL` • `NONE` • `0, 1, 2, ..., 4095` • `4100, 4101, 4102, ..., 4300` • A comma-separated list of alphanumeric strings beginning with letters. Underscores are also allowed. Spaces are not allowed.	• All trace events are caught. • All user trace events are caught. • All kernel trace events are caught. • No trace events are caught. • Listed user trace events are caught. • Listed kernel trace events are caught. • The tags of trace events as specified in an event-map file are caught. See “Understanding Event-Map Files” on page 5-10 for more information.
If-Expression	Boolean expression	Expression is any valid boolean C-like expression, possibly containing functions or macros. See Chapter 9 for more information.
Then-Expression	Numeric expression or string	Expression is any valid C-like expression, possibly containing functions or macros. See Chapter 9 for more information.
CPU List	`ALL`	All CPUs are listened to.
	`NONE`	No CPUs are listened to.
	`1, 2, 3, ...`	Listed CPUs are listened to.

Table 8-1. Common Configuration Parameters (Cont.)

Parameter Name	Possible Values	Meaning
PID List	Any meaningful combination of the following: • `ALL` • `NONE` • `123’1, 456’1, 789’1, ...` • A comma-separated list of alphanumeric strings beginning with letters. Underscores are also allowed. Spaces are not allowed.	 • All PIDs are listened to. • No PIDs are listened to. • Listed PIDs are listened to. • The name of a process.
TID List	Any meaningful combination of the following: • `ALL` • `NONE` • `123’1, 456’1, 789’1, ...` • A comma-separated list of alphanumeric strings beginning with letters. Underscores are also allowed. Spaces are not allowed.	 • All TIDs are listened to. • No TIDs are listened to. • Listed TIDs are listened to. • The name of a thread as specified in the `trace_open_thread()` call. See “trace_open_thread()” on page 3-9 for more information.
Node List	Any meaningful combination of the following: • `ALL` • `NONE` • `0, 1, 4` • A comma-separated list of host names. Spaces are not allowed.	 • All nodes are listened to. • No nodes are listened to. • Listed node IDs are listened to. • The name of a node/host.
Foreground Color	The colors your X server supports, as specified in the **`rgb.txt`** file.	The color used by the display object to draw text and graphics in the foreground.
Background Color	The colors your X server supports as, specified in the **`rgb.txt`** file.	The color in the background that any text and graphics are drawn over.
Font	The fonts your X server supports or are installed are in the directory **`/usr/lib/X11/fonts`**.	The style of text characters that the display object uses to display text.
Text Justify	Left	Text is justified on the left side of the display object.
	Center	Text is horizontally centered in the display object.
	Right	Text is justified on the right side of the display object.
	Default	Same as `Left`, unless a different default is specified in an X resource.

Table 8-1. Common Configuration Parameters (Cont.)

Parameter Name	Possible Values	Meaning
Text Gravity	Bottom	Text sinks to the bottom of the display object.
	Center	Text is vertically centered in the display object.
	Top	Text floats to the top of the display object.
	Default	Same as Bottom, unless a different default is specified in an X resource.

Display Object Name

The "Display Object Name" is the field at the top of the configuration form. This field is not titled in the configuration form; instead, it is labeled with the name of a display object type, for example, DataBox. This parameter allows:

- You to name a particular display object configuration. (By default, newly created display objects bear the name `unnamed_object`.)
- You to later define X resources to apply to the named display object. See Appendix B for more information.
- **ntrace** to reference the display object by name in error messages.

Event List

The Event List parameter restricts the trace events on which the display object can display information. The display object ignores any trace event IDs or trace event tags that are not on the trace event list. If an explicit list of trace event tags and trace event IDs is specified, the tags and IDs on the list must be separated by commas. Only listed trace events are examined. Qualified events and qualified states must not appear in the list.

If-Expression

The If-Expression parameter determines whether the Then-Expression parameter is evaluated. If-Expressions are boolean, i.e., they should evaluate to `false` (`0`) or `true` (non-zero). If the If-Expression is true, the Then-Expression is evaluated and displayed in the display object (assuming all other criteria are met). If an If-Expression evaluates to false, the Then-Expression retains its last value. See Chapter 9 for more information on expressions. Some examples of valid `If-Expressions` and their effect on the `Then-Expression` are shown in Table 8-2.

Table 8-2. Examples of If-Expressions

`If-Expression`	Effect on `Then-Expression`
`true`	Always evaluated
`false`	Never evaluated
`id() == 200`	Evaluated if current trace event ID is equal to 200
`id() < 200`	Evaluated if current trace event ID is less than 200
`pid() == 237’1`	Evaluated if current global process ID is equal to 237’1
`tid() == 895’3`	Evaluated if current NightTrace thread ID is equal to 895’3
`cpu() == 2 || cpu() == 4`	Evaluated if current trace event occurred on CPU 2 or 4

Then-Expression

The `Then-Expression` parameter determines what the output of the display object is when the `If-Expression` is true. If the `If-Expression` is false, the `Then-Expression` retains its last value. The possible values are a numeric expression or string. See Chapter 9 for more information on expressions. Some examples of valid `Then-Expressions` and their resulting values are shown in Table 8-3.

Table 8-3. Examples of Then-Expressions

`Then-Expression`	Resulting Value or Meaning
`id()`	The current trace event ID
`arg2()`	The second argument of the current trace event
`format (“abc=%d”, arg1())`	The string `“abc=10”` if `arg1()` is 10 (See “format()” on page 9-80.)
`get_string (curr_state, id())`	The string from the `curr_state` string table pointed to by `id()` (if any)
`get_string (event, id())`	Depending on whether trace event ID returned by `id()` is in the pre-defined `event` table, either the trace event ID number or its corresponding trace event tag is displayed. (See “get_string()” on page 9-75, “Pre-Defined String Tables” on page 5-15, and “id()” on page 9-19.)
`get_string (pid, pid())`	Depending on whether the global process identifier returned by `pid()` is in the pre-defined `pid` table, either the global process identifier (PID) or its corresponding process name is displayed. (See “get_string()” on page 9-75, “Pre-Defined String Tables” on page 5-15, and “pid()” on page 9-22.)

Table 8-3. Examples of Then-Expressions (Cont.)

Then-Expression	Resulting Value or Meaning
`get_string (tid, tid())`	Depending on whether the NightTrace thread identifier returned by `tid()` is in the pre-defined `tid` table, either the NightTrace thread identifier (TID) or its corresponding thread name is displayed. (See “get_string()” on page 9-75, “Pre-Defined String Tables” on page 5-15, and “tid()” on page 9-25.)
`get_string (boolean,` *arg*`)`	If *arg* has the value 0, `false` is displayed. Otherwise, `true` is displayed. (See “get_string()” on page 9-75 and “Pre-Defined String Tables” on page 5-15)
`get_string (syscall,` *arg*`)`	*arg*’s value is looked up in the pre-defined `syscall` table, and its corresponding system call name is displayed. (This is meaningful only for NightTrace kernel trace event files.) (See “get_string()” on page 9-75 and “Kernel String Tables” on page 11-32.)
`get_string (vector,` *arg*`)`	*arg*’s value is looked up in the pre-defined `vector` table, and its corresponding interrupt or exception name is displayed. (This is meaningful only for NightTrace kernel trace event files.) (See “get_string()” on page 9-75 and “Kernel String Tables” on page 11-32.)
`get_format (next_state, id())`	The formatted string from the `next_state` format table indexed by the integer returned by `id()` (if any)
`get_format (state_summary)`	Display statistics about state matches, the state gaps, and the state durations. (See “get_format()” on page 9-79 and “Pre-Defined Format Tables” on page 5-21.)
`get_format(event_summary)`	Display statistics about trace event matches and trace event gaps. (See “get_format()” on page 9-79 and “Pre-Defined Format Tables” on page 5-21.)
`get_format(event_arg_summary,3)`	Display statistics about trace event matches and their type long third argument. (See “get_format()” on page 9-79 and “Pre-Defined Format Tables” on page 5-21.)
`get_format(event_arg_dbl_summary,1)`	Display statistics about trace event matches and their type double first argument. (See “get_format()” on page 9-79 and “Pre-Defined Format Tables” on page 5-21.)

CPU List

The `CPU List` parameter determines from which logical central processing units (CPUs) the display object will process trace events. Only processes that run on one of the CPUs on this list will be considered by this display object. If the trace event sent to the display object is not on the list of CPUs, then the trace event is ignored. A CPU number can be specified only if a NightTrace kernel trace event file is specified. Multiple CPU numbers must be separated by commas.

PID List

A *global process identifier* (PID) is a 32-bit integer. It includes a 16-bit integer *raw PID* and a 16-bit integer *lightweight process identifie*r (LWPID). The syntax for specifying a PID is:

raw_PID'LWPID

The `PID List` parameter is the list of global process identifiers (PIDs) or process names that the display object will accept trace events from. If the trace event did not occur in a process listed in this parameter, the trace event is ignored. If a number or name is specified that is not a valid PID, a warning message is displayed. Multiple numbers and names must be separated by commas.

NOTE

Prior to Version 4.1, **`ntrace`** converted process identifiers into process names during `PID List` input verification for a display object. For each process identifier in the `PID List`, **`ntrace`** would try to find its associated process name and display that name in the `PID List`. However, because multiple processes having the same name may exist on a system, changing a process identifier into a process name introduces the possibility that the display object will accept trace events from undesirable processes. Therefore, **`ntrace`** no longer performs this conversion.

For example, suppose that two processes named **`a.out`** exist on a particular system and that one has a PID of `1234` and the other has a PID of `5678`. Further suppose that you wish to create a StateGraph to display events only for PID `1234`. Prior to Version 4.1, if you entered `1234` in the `PID List` parameter, **`ntrace`** would have converted that to **`a.out`**. As the events were being analyzed, any event that had a PID of `5678` would also have been displayed by the StateGraph since a process named **`a.out`** also existed with a PID of `5678`.

If the trace event file has multiple processes with the same name (for example, `a.out`), specifying any one of the PIDs for that process selects all the PIDs of that process. To avoid this, it is recommended that all processes be given unique names. If that is not possible, you can isolate individual processes by including a PID restriction in the `If-Expression` parameter. For example, if `a.out` includes PIDs 100’1, 200’1, and 300’1 and you want information only on PID 100’1, set the `PID List` parameter to `a.out` and the `If-Expression` to `pid() == 100’1`. For more information about the `pid` function, see “pid()” on page 9-22.

TID List

A *NightTrace thread identifier* (TID) is a 32-bit integer. It includes a 16-bit integer *raw PID* and a 16-bit integer *C thread* or *Ada task identifier*. If neither C threads nor Ada tasks are in use, then the 16-bit integer will always be zero. The syntax for specifying a TID is:

raw_PID'*task_id*

or:

raw_PID'*thread_id*

The `TID List` parameter is the list of NightTrace thread identifiers (TIDs) or thread names that the display object will accept trace events from. If the trace event did not occur in a thread listed in this parameter, the trace event is ignored. If a number or name is specified that is not a valid TID, a warning message is displayed. Multiple numbers and names must be separated by commas.

NOTE

Prior to Version 4.1, **`ntrace`** converted thread identifiers into thread names during `TID List` input verification for a display object. For each thread identifier in the `TID List`, **`ntrace`** would try to find its associated thread name and display that name in the `TID List`. However, because multiple threads having the same name may exist on a system, changing a thread identifier into a thread name introduces the possibility that the display object will accept trace events from undesirable threads. Therefore, **`ntrace`** no longer performs this conversion.

For example, suppose that two threads named `daemon` exist on a particular system and that one has a TID of `1234'1` and the other has a TID of `5678'3`. Further suppose that you wish to create a StateGraph to display events only for TID `1234'1`. Prior to Version 4.1, if you entered `1234'1` in the `TID List` parameter, **`ntrace`** would have converted that to `daemon`. As the events were being analyzed, any event that had a TID of `5678'3` would also have been displayed by the StateGraph since the thread `daemon` also existed with a TID of `5678'3`.

If the trace event file has multiple threads with the same name (for example, `CHILD_THREAD`), specifying any one of the TIDs with that thread name selects all of the TIDs with that thread name. To avoid this, it is recommended that all threads be given unique names. If that is not possible, you can isolate individual threads by including a TID restriction in the `If-Expression` parameter. For example, if `CHILD_THREAD` includes TIDs 100’1, 200’1, and 300’1 and you want information only on TID 100’1, set the `TID List` parameter to `CHILD_THREAD` and the `If-Expression` to `tid() == 100'1`. For more information on thread names, see “trace_open_thread()” on page 3-9. For more information about the `tid` function, see“tid()” on page 9-25.

Node List

When NightTrace processes a trace file which was timestamped by an RCIM synchronized tick clock, it internally assigns a node identifier to each node/host name represented by a trace file. If no trace file was generated using the tick clock, this parameter is not displayed.

The Node List parameter is the list of node identifiers or node names from which the display object will accept trace events. If the trace event did not occur on a node listed in this parameter, the trace event is ignored. If a number or name is specified that is not a valid node, a warning message is displayed. Multiple numbers and names must be separated by commas.

Foreground Color

The Foreground Color parameter determines the color of items in the foreground of the display object, which usually corresponds to the data being displayed by the display object.

Background Color

The Background Color parameter determines the color of the background of the display object. Although this is not the color used to display the data of interest in the display object, it should be a color that contrasts well with the Foreground Color. This will make the data easier to read.

Font

The Font parameter determines the font that characters in the display object are displayed in. Use of a small font size is recommended due to the fact that there is generally a lot of data being displayed and a small font size will help conserve screen space. All examples in this manual use the default "`fixed`" font that is supplied with all X servers.

Text Justify

The Text Justify parameter determines the justification of the text in the display object. Figure 8-2 shows what each type of text-justification looks like.

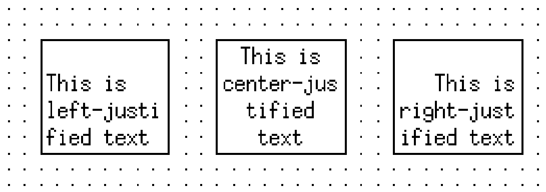

This is left-justi fied text
This is center-jus tified text
This is right-just ified text

Figure 8-2. Left-, Center-, and Right-Justified Text

Text Gravity

The Text Gravity parameter determines whether text in the object will float to the top or sink to the bottom of the display object. Figure 8-3 shows what each type of text gravity looks like.

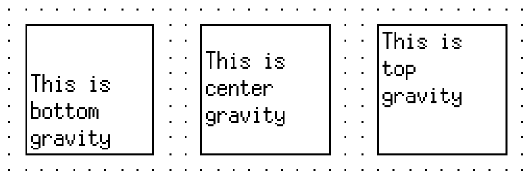


Figure 8-3. Top vs. Bottom Gravity

Configuration Form Push Buttons

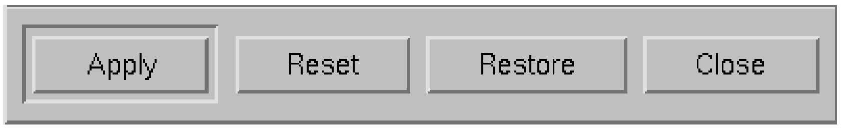


Figure 8-4. Configuration Form Push Buttons

Figure 8-4 shows the push buttons that all display object configuration forms have.

After you have changed the configuration parameters of a display object, these buttons allow you to perform the following operations:

Apply	(default) Validate the changes you made to the configuration parameters, and apply the changes to the display object. This is equivalent to pressing <Enter>.

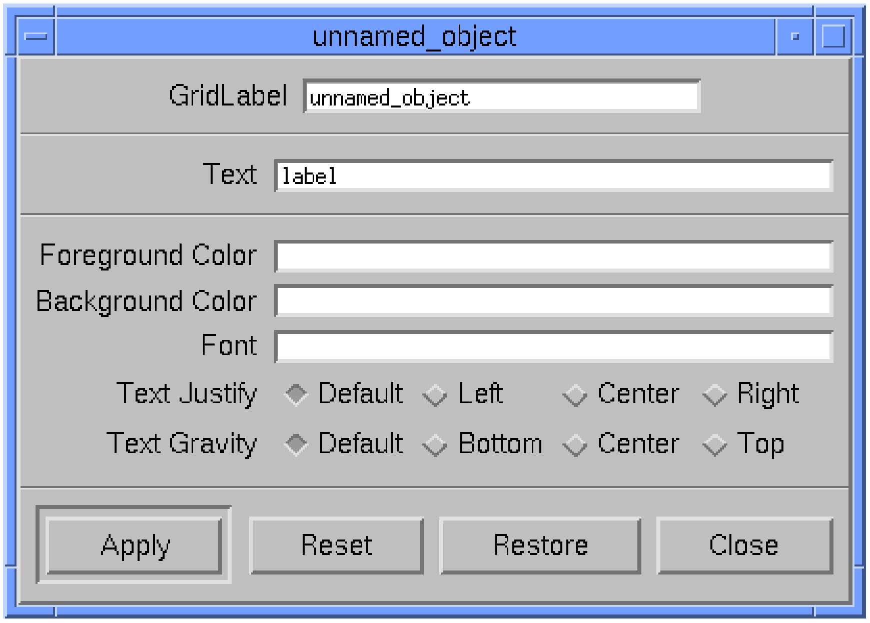
Reset	Discard all changes made since the last Apply or <Enter>. This is equivalent to pressing <Esc>.
Restore	Discard all changes made since the window was opened.
Close	Discard any changes made since the last change was applied and close the window.

Specific Configuration Parameters

The following sections discuss the configuration parameters specific to the following display objects:

- GridLabel
- DataBox
- StateGraph
- EventGraph
- DataGraph
- Ruler

GridLabel



GridLabel: unnamed_object

Text: label

Foreground Color:

Background Color:

Font:

Text Justify: [✓ Default] Left Center Right

Text Gravity: [✓ Default] Bottom Center Top

Figure 8-5. GridLabel Configuration Form

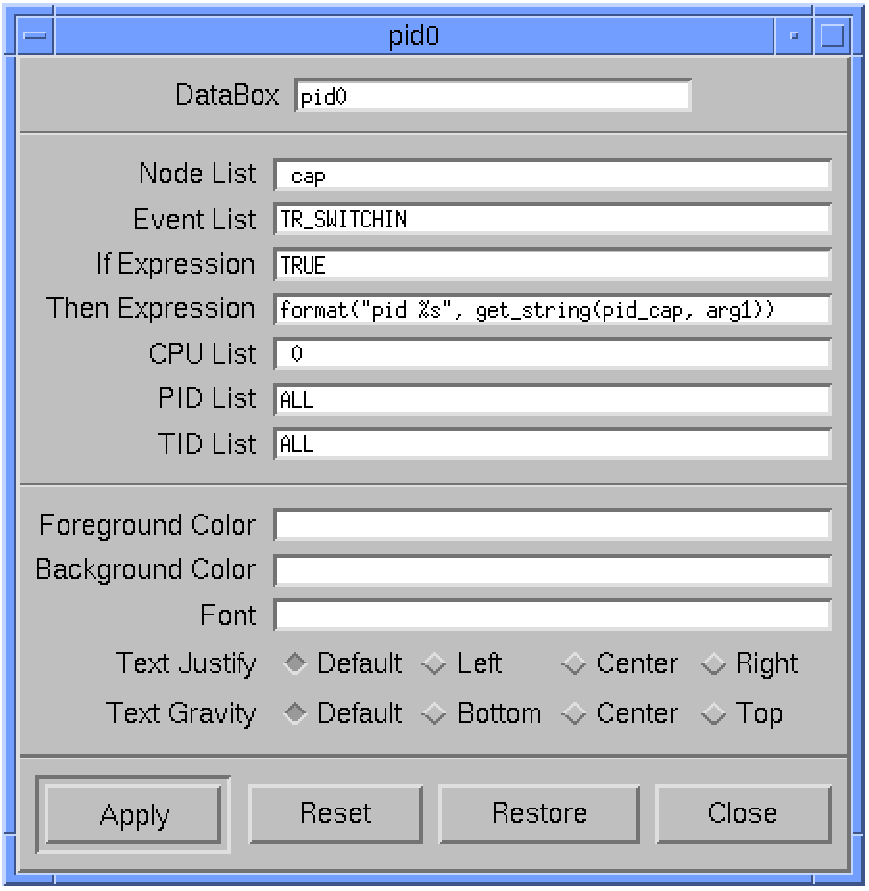
The configuration form for the GridLabel is shown in Figure 8-5.

The Text parameter is the only parameter that is unique to GridLabels. This parameter is set to the characters that are to appear in the GridLabel. For example, if you want a box on the grid containing the phrase, “Flight Simulation Trace Screen,” you would enter the following text in the Text field:

```
Flight Simulation Trace Screen
```

See “GridLabel” on page 7-12 for more information. See “Common Configuration Parameters” on page 8-1 for descriptions of the common configuration parameters that GridLabels use.

DataBox



DataBox	pid0
Node List	cap
Event List	TR_SWITCHIN
If Expression	TRUE
Then Expression	format("pid %s", get_string(pid_cap, arg1))
CPU List	0
PID List	ALL
TID List	ALL
Foreground Color	
Background Color	
Font	
Text Justify	[✓ Default] Left Center Right
Text Gravity	[✓ Default] Bottom Center Top

Figure 8-6. DataBox Configuration Form

The configuration form for the DataBox is shown in Figure 8-6.

NOTE

The Node List field appears in this dialog only when NightTrace is configured to use an RCIM to timestamp events.

A DataBox can be used as a counter. A counter is simply a DataBox that counts the occurrences of a particular trace event or other condition up to the current time.

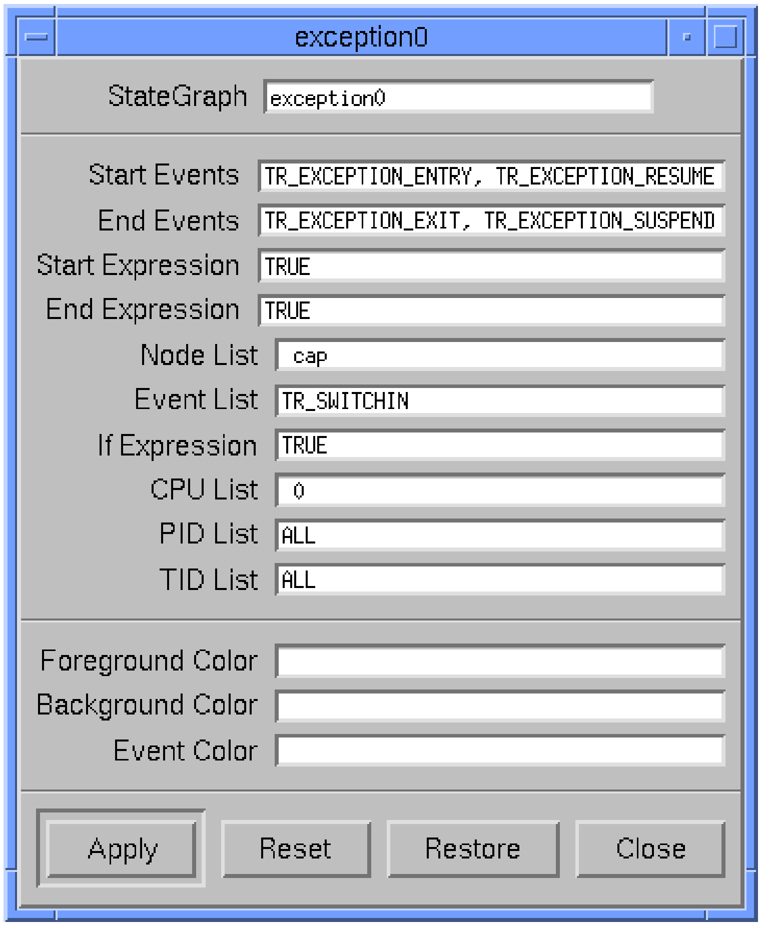
For example, if you wanted to display the number of trace events occurring before the current time, set the Event List parameter to ALL and put the following expression in the Then-Expression field:

```
event_matches()
```

This expression counts the number of times the criteria were met. See Chapter 9 for more information on expressions. See "DataBox" on page 7-12 for more information. See "Common Configuration Parameters" on page 8-1 for descriptions of the common configuration parameters that DataBoxes use.

To determine the format of the data displayed in the DataBox, give the Then-Expression parameter an expression value. See "Then-Expression" on page 8-5 for examples.

StateGraph


exception0

StateGraph	exception0
Start Events	TR_EXCEPTION_ENTRY, TR_EXCEPTION_RESUME
End Events	TR_EXCEPTION_EXIT, TR_EXCEPTION_SUSPEND
Start Expression	TRUE
End Expression	TRUE
Node List	cap
Event List	TR_SWITCHIN
If Expression	TRUE
CPU List	0
PID List	ALL
TID List	ALL
Foreground Color	
Background Color	
Event Color	

Figure 8-7. StateGraph Configuration Form

The configuration form for the StateGraph is shown in Figure 8-7.

NOTE

The Node List field appears in this dialog only when NightTrace is configured to use an RCIM to timestamp events.

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.

The Start-Events parameter determines the trace events that can begin a state. This parameter, along with the End-Events parameter, defines part of what will be considered a state for this display object. These parameters work exactly like the Event List parameters discussed earlier in “Common Configuration Parameters” on page 8-1. Like the Event List, they each have corresponding If-Expressions, called Start-Expression and End-Expression, respectively.

The Start-Expression parameter determines the criteria, in addition to the start trace event(s) and other criteria, which must be true before a state is considered to be started (active). The End-Expression parameter determines the criteria, in addition to the end trace event(s) and other criteria, which must be true before a state is considered to be ended (inactive).

The following semantic rules apply to these expressions. In these rules, *defining state* means a state with trace events in the Start-Events and End-Events lists.

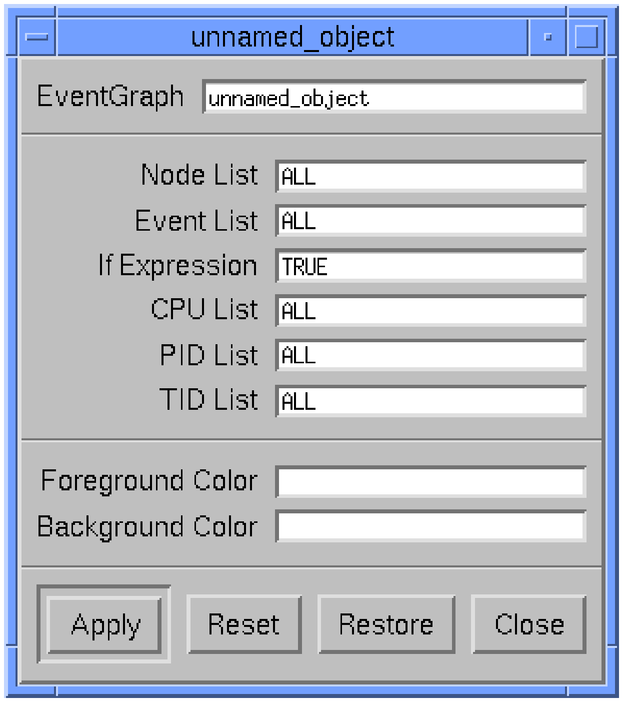
- Start-Expression must not refer to its defining states. For example, it must not call `state_dur()`, `state_gap()`, start or end functions for these states. (See “Multi-State Functions” on page 9-56, “Start Functions” on page 9-34, and “End Functions” on page 9-45 for details.) Calling these functions for these states would be an attempt to define a state based on its own definition. Note that Start-Expression may call all of these functions for qualified states.
- End-Expression must not refer to its defining states. For example, it must not call `state_dur()`, `state_gap()`, or end functions for these states. Calling these functions for these states would be an attempt to define a state based on its own definition. Note that End-Expression may call start functions for these states because at this point in the state definition, the state has started. Note also that End-Expression may call all of these functions for qualified states.

The Event Color parameter specifies the color of the vertical lines that represent the events in the Event List. The possible values are the colors your X server supports, as specified in the **`rgb.txt`** file. See Appendix B for more information.

StateGraphs indicate when a state is active by drawing a rectangle in the Foreground Color that spans the time when the start state and end state criteria are met. In addition to drawing this state rectangle, StateGraphs can behave exactly like EventGraphs by using the Event List and If-Expression fields. 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.

See “StateGraph” on page 7-14 for more information. See “Common Configuration Parameters” on page 8-1 for descriptions of the common configuration parameters that StateGraphs use.

EventGraph


unnamed_object

EventGraph: unnamed_object

Node List: ALL
Event List: ALL
If Expression: TRUE
CPU List: ALL
PID List: ALL
TID List: ALL

Foreground Color:
Background Color:

Figure 8-8. EventGraph Configuration Form

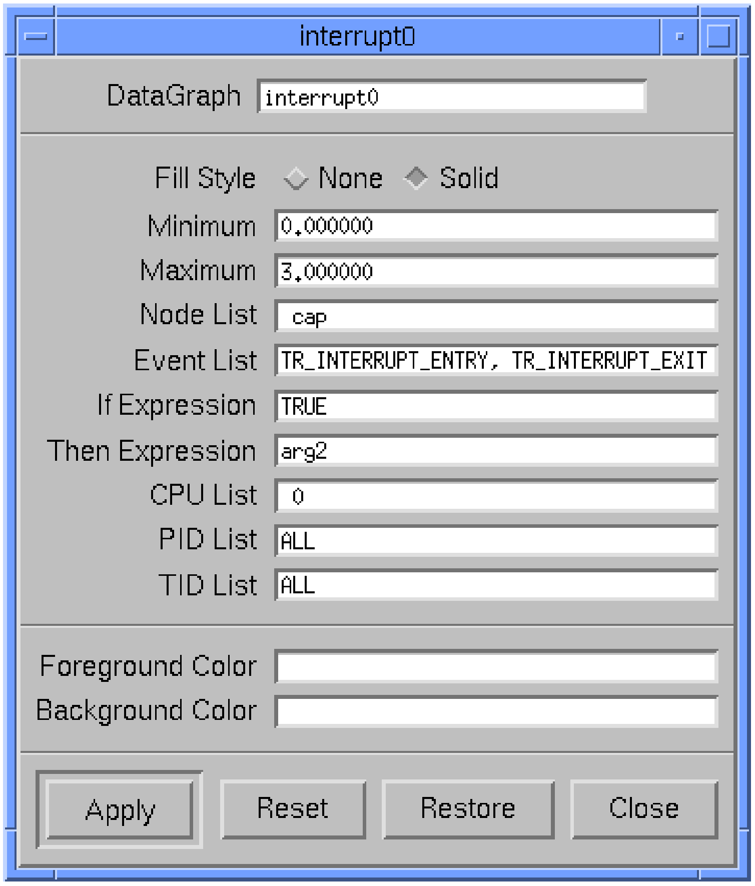
The configuration form for the EventGraph is shown in Figure 8-8. All of the parameters for the EventGraph are discussed in “Common Configuration Parameters” on page 8-1. See “EventGraph” on page 7-15 for more information.

NOTE

The Node List field appears in this dialog only when NightTrace is configured to use an RCIM to timestamp events.

The If-Expression of an EventGraph determines whether a trace event should be graphed. If the If-Expression is true, then a vertical line is drawn at the point in time that the trace event occurred.

DataGraph



interrupt0

DataGraph	interrupt0
Fill Style	None [✓ Solid]
Minimum	0.000000
Maximum	3.000000
Node List	cap
Event List	TR_INTERRUPT_ENTRY, TR_INTERRUPT_EXIT
If Expression	TRUE
Then Expression	arg2
CPU List	0
PID List	ALL
TID List	ALL
Foreground Color	
Background Color	

Figure 8-9. DataGraph Configuration Form

The configuration form for the DataGraph is shown in Figure 8-9.

NOTE

The Node List field appears in this dialog only when NightTrace is configured to use an RCIM to timestamp events.

The Fill Style parameter determines the style of DataGraph created. The possible choices are None or Solid. If None is chosen, then a vertical line is drawn only at the time of a trace event. If Solid is chosen, then all space to the right of a trace event will be filled until the next trace event is encountered. Figure 8-10 shows the difference between Solid and None.

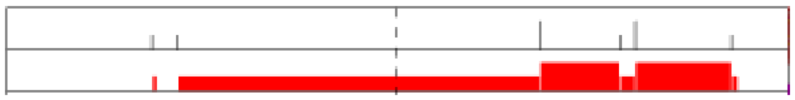


Figure 8-10. Solid vs. No Fill

The `Maximum` parameter determines what data value corresponds to the top of the DataGraph. 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 DataGraph. 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.

The `Minimum` parameter determines what data value corresponds to the bottom of the DataGraph. 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 DataGraph. 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.

Figure 8-11 shows the same set of data drawn in three DataGraphs, each configured differently. The data range in value from 1 to 6 and are shown at the bottom of the figure.

- The top DataGraph is configured with a minimum of 2 and a maximum of 4. Notice that several bars reach the top of the DataGraph 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 DataGraph is configured with a minimum of 0 and a maximum of 10. Notice that the bars do not reach the top of the DataGraph and that the differences between values are harder to discern.
- The bottom DataGraph 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 DataGraph.

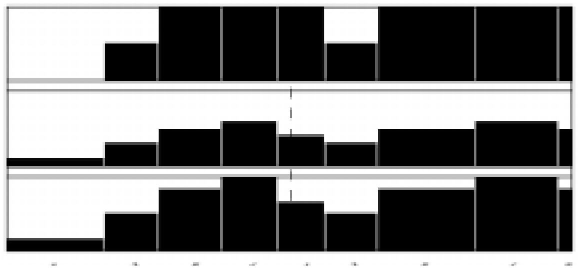
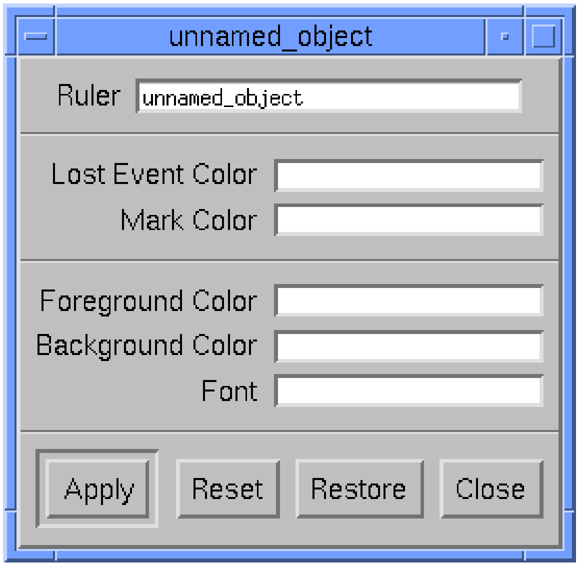


Figure 8-11. Maximum vs. Minimum Values

See “DataGraph” on page 7-16 for more information. See “Common Configuration Parameters” on page 8-1 for descriptions of the common configuration parameters that DataGraphs use.

Ruler



Ruler	unnamed_object
Lost Event Color	
Mark Color	
Foreground Color	
Background Color	
Font	

Figure 8-12. Ruler Configuration Form

The configuration form for the Ruler is shown in Figure 8-12.

The Lost Event Color parameter specifies the color of the reverse-video “L” (shown in Figure 8-13) that is placed on a Ruler where NightTrace lost data. The possible values are the colors your X server supports, as specified in the **rgb.txt** file. See “Preventing Trace Events Loss” on page A-1 for more information on lost data.

The Mark Color parameter specifies the color of the mark indicator, a triangle that appears on the Ruler (shown in Figure 8-13). The possible values are the colors your X server supports. See “The Interval Push Buttons” on page 6-8 for more information about the mark.

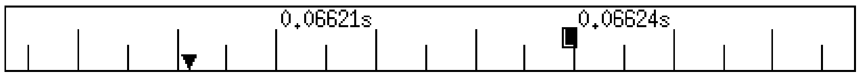


Figure 8-13. Mark and Lost Event Markers

See “Ruler” on page 7-17 for more information. See “Common Configuration Parameters” on page 8-1 for descriptions of the common configuration parameters that Rulers use.

9
Using Expressions

9
Using Expressions

Overview

NightTrace allows you to define macros, qualified events, and qualified states to aid in the analysis of trace data. *Macros* are named expressions provided for flexibility and convenience. *Qualified events* provide a mechanism for referencing *trace event configurations* within certain *functions*. *Qualified states* provide a mechanism for referencing *state configurations* within certain functions as well.

The Expressions menu contains menu items for creating these entities. See “Expressions Menu” on page 9-1, “Macros” on page 9-6, “Qualified Events” on page 9-81, and “Qualified States” on page 9-83 for further information.

Macros, qualified events, and qualified states are configured using *expressions* in much the same way as *display objects*. See “Expressions” on page 9-4 for a complete explanation of expressions. In addition, Chapter 8 - Configuring Display Objects may provide some helpful information as well.

Expressions Menu

Figure 9-1 shows the display page menu that lets you define macros, qualified events, and qualified states. For more information about display pages, see “Understanding Page Configuration Files” on page 5-12.

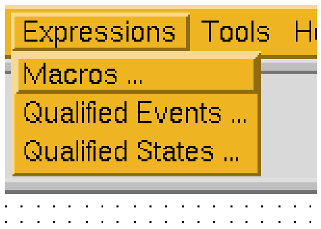


Figure 9-1. Expressions Menu

Selecting any of these menu entries makes an Expression Dialog Box appear.

Expression Dialog Boxes

In the following text, *expr* stands for macro, qualified event, and qualified state.

Selecting any of the entries from the Expressions menu of the display page, makes a dialog box like the one in Figure 9-2 appear. Because all *exprs* are user-defined, the list of *exprs* is empty at first.

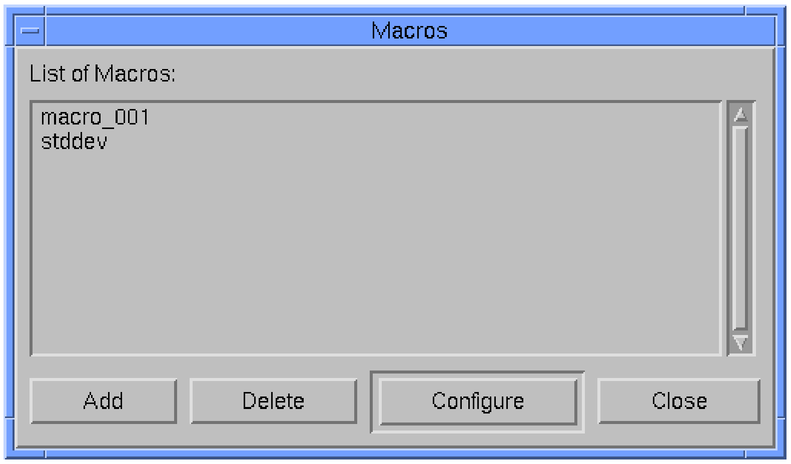


Figure 9-2. Macro Dialog Box

The caption and the list presented are suitably different for each of the *expr* dialog boxes.

The push buttons in the dialog boxes perform the following functions:

Add	Create a new *expr* on the current display page. The initial name of an *expr* is *type_###*, where *type* is `macro`, `event`, or `state` and *###* is a three-digit number beginning with `001`.
Delete	Remove the selected *expr*
Configure	(default) Reconfigure or rename the selected *expr*
Close	Close the dialog box

Add, Delete, and Close need no further explanation. Selecting Configure makes an *expr* Configuration Form appear.

Expression Configuration Forms

In the following text, *expr* stands for macro, qualified event, and qualified state.

The Configuration Forms for *exprs* are similar. Common features are described here and specific features appear in later sections.

The push buttons on a Configuration Form appear in Figure 9-3.

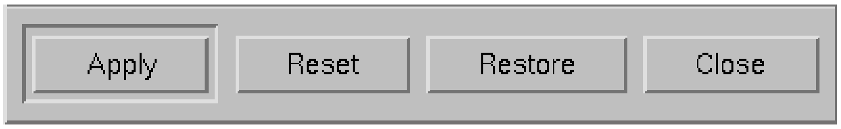


Figure 9-3. Configuration Form Push Buttons

A description of these push buttons follows:

Apply	(default) Validate the changes you made to the configuration parameters, and apply the changes to the selected *expr*. This is equivalent to pressing <Enter>.
Reset	Discard all changes made since the last Apply or <Enter>. This is equivalent to pressing <Esc>.
Restore	Discard all changes made since the window was opened.
Close	Discard any changes made since the last change was applied and close the window.

When you have finished editing the fields on the Configuration Form, press <Enter> or click on Apply. This causes NightTrace to validate the data in each field you modified. For general information on field editing and how NightTrace handles editing errors, see “Field Editing” on page 6-16.

exprs are saved in a configuration file but are global to all display pages. That is, if an *expr* is created in one display page, it may be used by any other display page. This means, however, that if an *expr* is saved in one configuration file but altered in another, you will have to reopen the file with the original copy of the *expr* and save the new value.

NightTrace prevents you from creating more than one definition for a specific *expr*. If you wish to change the definintion of an *expr*, you must select it from the list of *exprs* and press Configure. See “Expression Dialog Boxes” on page 9-2 for details.

TIP:

If you want to share *exprs* among multiple display pages, create an empty display page and put only *exprs* in it. Any new *exprs* or changes to old *exprs* should be added to this display page. It is also a good idea to place a DataBox on this page for every *expr* that you add to this page. This way, you can see the current value of all your *exprs* at a glance.

Expressions

NightTrace expressions can evaluate to numbers, strings, or boolean values. Expressions appear in the following places in NightTrace:

- Start-Expression and End-Expression on:
  - Configuration Forms
  - Summarize Forms
- If-Expression on:
  - Configuration Forms
  - Summarize Forms
  - Search Forms
- Then-Expression on Configuration Forms
- Filter-Expression and Summary-Expression on the Summarize Form
- Expression on Macro Dialog Boxes
- Values in format tables
- Calls to `format()`, `get_string()`, `get_item()`, `get_format()`, and summary functions.

Start-Expressions, End-Expressions, If-Expressions, and Filter-Expression must evaluate to boolean values.

See Chapter 8 for more information on the Configuration Form. See Chapter 10 for more information on the Search and Summarize Forms. See “Format Tables” on page 5-18 for more information on format tables. Information on `format()`, `get_string()`, `get_item()`, `get_format()`, and summary functions appears later in this chapter.

NightTrace expressions are comprised of a combination of operators and operands. A description of these operators and operands appears in the following sections.

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

Operand types are largely based on the C programming language and include:

- integer
- double-precision floating point
- character
- string
- boolean

Operands include:

- constants (see “Constants” on page 9-5)
- macro calls (see “Macros” on page 9-6)
- function calls (see “Functions” on page 9-9)
- qualified events (in functions only) (see “Qualified Events” on page 9-81)
- qualified states (in functions only) (see “Qualified States” on page 9-83)

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 9-1 shows units and suffixes for time constants.

Table 9-1. 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`

Macros

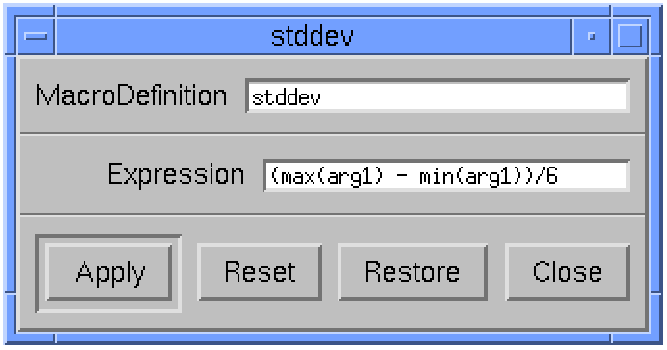
Macros are named expressions provided for flexibility and convenience. Table 9-2 contrasts functions and macros.

Table 9-2. A Comparison of Functions and Macros

Functions	Macros
Predefined	User-defined
May have parameters	Cannot have parameters
Invoked with parentheses around the parameter list	Invoked with a dollar sign ($) before the macro name

To create a macro definition, select the Macros menu item from the Expressions menu (see “Expressions Menu” on page 9-1) to open the Macro Dialog Box (see “Expression Dialog Boxes” on page 9-2 for details on this type of dialog).

Click the Add button on the Macro Dialog Box, select the macro from the list, and click on the Configure button to pop up a Macro Configuration Form, like the one shown in Figure 9-4.



MacroDefinition stddev

Expression (max(arg1) - min(arg1))/6

Figure 9-4. Macro Configuration Form

The following parameters can be configured for a macro.

MacroDefinition The name by which you refer to this macro in expressions. Only references to this macro have a dollar sign ($) prefix.

Expression Any valid expression. You must not call macros recursively; if you try it, NightTrace issues an error, and you get undefined results. Macros must not call the `format()` and `get_format()` functions. (For more information about these functions, see "format()" on page 9-80 and "get_format()" on page 9-79.)

EXAMPLES

A StateGraph configuration is a good candidate for a macro because it has two expressions that are often related. For example, the following configuration

```
Start Events:     FOO
Start Expression: arg1() == 0x1234 &&
    (arg2() == 0 || arg3() > 700)
End Events:       BAR
End Expression:   arg1() == 0x1234 &&
    (arg2() == 0 || arg3() > 700)
```

graphs states of trace event FOO through trace event BAR, where the arguments of the trace events must meet an identical criteria to be considered interesting. Making

```
arg1() == 0x1234 && (arg2() == 0 || arg3() > 700)
```

a macro would help ensure that you did not type the expression wrong in one of the fields, and it would allow you to change the expressions easily, even while viewing the trace run in View mode. (You can leave Macro Configuration Forms up while in View mode.)

Another good use for a macro is for focusing many display objects on a specific process group. For example, if a Column contained several EventGraphs, each of which had the following If-Expression:

```
If Expression: process_name() == $task
```

then a `task` macro definition of

```
“foobar”
```

would cause all of the EventGraphs to show only trace events logged by process `foobar`. Changing the macro to

```
“bazonk”
```

would shift the focus of the EventGraphs from process `foobar` to process `bazonk`. This technique can also be used in DataBoxes, DataGraphs, and StateGraphs.

Functions

Functions are pre-defined NightTrace entities that may be used in an *expression*. Night-Trace defines five classes of functions:

- Trace event functions (see “Trace Event Functions” on page 9-19)
- State functions (see “State Functions” on page 9-34)
- Offset functions (see “Offset Functions” on page 9-59)
- Summary functions (see “Summary Functions” on page 9-70)
- Format and table functions (see “Format and Table Functions” on page 9-75)

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 9-59.
`start_`	Functions with this prefix provide information about the start *event* of the *most recent instance of a state.* See “Start Functions” on page 9-34.
`end_`	Functions with this prefix provide information about the end *event* of the *last completed instance of a state* See “End Functions” on page 9-45.
`state_`	Functions with this prefix provide information about instances of states. See “Multi-State Functions” on page 9-56.
`event_`	Functions with this prefix provide information about instances of events. See “Multi-Event Functions” on page 9-32.

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 9-3 contains a complete list of functions.

Table 9-3. NightTrace Functions

Syntax	Return Type
`id` [([*QE*])] `start_id` [([*QS*])] `end_id` [([*QS*])] `offset_id` (*offset_expr*)	The integer *trace event ID*.
`arg`[*N*] [([*QE*])] `start_arg`[*N*] [([*QS*])] `end_arg`[*N*] [([*QS*])] `offset_arg`[*N*] (*offset_expr*)	The integer *trace event argument*.
`arg`[*N*]`_dbl` [([*QE*])] `start_arg`[*N*]`_dbl` [([*QS*])] `end_arg`[*N*]`_dbl` [([*QS*])] `offset_arg`[*N*]`_dbl` (*offset_expr*)	The double-precision floating point *trace event argument*.
`num_args` [([*QE*])] `start_num_args` [([*QS*])] `end_num_args` [([*QS*])] `offset_num_args` (*offset_expr*)	The number of arguments associated with a *trace event*.
`pid` [([*QE*])] `start_pid` [([*QS*])] `end_pid` [([*QS*])] `offset_pid` (*offset_expr*)	The integer global process identifier (*PID*) associated with a *trace event*.
`raw_pid` [([*QE*])] `start_raw_pid` [([*QS*])] `end_raw_pid` [([*QS*])] `offset_raw_pid` (*offset_expr*)	The integer process identifier (*raw PID*) associated with a *trace event*.
`lwpid` [([*QE*])] `start_lwpid` [([*QS*])] `end_lwpid` [([*QS*])] `offset_lwpid` (*offset_expr*)	The integer lightweight process identifier (*LWPID*) associated with a *trace event.*
`thread_id` [([*QE*])] `start_thread_id` [([*QS*])] `end_thread_id` [([*QS*])] `offset_thread_id` (*offset_expr*)	The integer *thread* identifier (*thread ID*) associated with a *trace event*.
`task_id` [([*QE*])] `start_task_id` [([*QS*])] `end_task_id` [([*QS*])] `offset_task_id` (*offset_expr*)	The integer Ada task identifier associated with a *trace event*.
`tid` [([*QE*])] `start_tid` [([*QS*])] `end_tid` [([*QS*])] `offset_tid` (*offset_expr*)	The integer NightTrace thread identifier (*TID*) associated with a *trace event*.

Table 9-3. NightTrace Functions

Syntax	Return Type
`cpu` [([*QE*])] `start_cpu` [([*QS*])] `end_cpu` [([*QS*])] `offset_cpu` (*offset_expr*)	The integer logical CPU number associated with a *trace event*. This function is only valid when applied to events from Night-Trace kernel trace event files.
`time` [([*QE*])] `start_time` [([*QS*])] `end_time` [([*QS*])] `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` [([*QE*])] `start_node_id` [([*QS*])] `end_node_id` [([*QS*])] `offset_node_id` (*offset_expr*)	The internally-assigned integer *node identifier* associated with a *trace event*.
`pid_table_name` [([*QE*])] `start_pid_table_name` [([*QS*])] `end_pid_table_name` [([*QS*])] `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` [([*QE*])] `start_tid_table_name` [([*QS*])] `end_tid_table_name` [([*QS*])] `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` [([*QE*])] `start_node_name` [([*QS*])] `end_node_name` [([*QS*])] `offset_node_name` (*offset_expr*)	The string describing the name of the system from which a *trace event* was logged.
`process_name` [([*QE*])] `offset_process_name` (*offset_expr*)	The string describing the name of the process (*PID*) associated with a *trace event*.
`task_name` [([*QE*])] `offset_task_name` (*offset_expr*)	The string describing the name of the Ada *task* associated with a *trace event*.
`thread_name` [([*QE*])] `offset_thread_name` (*offset_expr*)	The string describing the name of the C *thread* associated with a *trace event*.
`event_gap` [([*QE*])] `state_gap` [([*QS*])]	The double-precision floating point time, expresed in units of seconds, between the instances of either a *trace event* or a *state*.
`state_dur` [([*QS*])]	The double-precision floating point time, expressed in units of seconds, of an instance of a *state*.
`event_matches` [([*QE*])] `state_matches` [([*QS*])] `summary_matches` [()]	The integer number of instances of either a *trace event* or a *state*.
`state_status` [([*QS*])]	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 9-58 for important details.

Table 9-3. NightTrace Functions

Syntax	Return Type
`offset` [([*QE*])] `start_offset` [([*QS*])] `end_offset` [([*QS*])]	The integer ordinal number (*offset*) of a *trace event*.
`min_offset` (*expr*) `max_offset` (*expr*)	The integer ordinal number (*offset*) of a *trace event* associated with a minimum or maximum occurrence of *expr.*
`min` (*expr*) `max` (*expr*) `avg` (*expr*) `sum` (*expr*)	The minimum, maximum, average, or sum of *expr* values before the *current time*. The return type is that of *expr.*
`get_string` (*table_name*[, *int_expr*])	The character string associated with item *int_expr* in string table *table_name.*
`get_item` (*table_name*, "*str_const*")	The first integer item number associated with string *str_const* in string table *table_name*.
`get_format` (*table_name*[, *int_expr*])	The character string associated with item *int_expr* in format table *table_name.*
`format` ("*format_string*" [, *arg*] ...)	A character string to format and display.

Function Parameters

If the function has a *parameter*, the parentheses are required. Otherwise, they are optional. For example,

`arg2`	No parentheses are required
`arg2()`	No parentheses are required
`arg2(GAK)`	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_event)`	Operates on the *qualified event* `my_event`
`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 *qualified state* `my_state`

This manual uses the following conventions for function *parameters*:

QE	A user-defined *qualified event*. If supplied, the function applies to the specified qualified event. For more information, see "Qualified Events" on page 9-81.
QS	A user-defined *qualified state*. If supplied, the function applies to the specified qualified state. For more information, see "Qualified States" on page 9-83.
offset_expr	An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.
expr	Any valid NightTrace *expression* (see "Expressions" on page 9-4).
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 *macros*, *qualified events*, and *qualified states*; it temporarily puts their names in the appropriate `Dialog Box` in case you want to configure these constructs. For more information about these `Dialog Boxes`, see "Expression Dialog Boxes" on page 9-2.

Function Terminology

In order to use the NightTrace functions effectively, it may be useful to understand some of the concepts associated with them.

Remember that an *event* (or *trace event*) is either a user-defined point of interest in an application's source code or a predefined point of interest in the kernel. In addition, a *state* is defined to be a region of source code bounded by two 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 9-5 illustrates some of these concepts with a StateGraph.

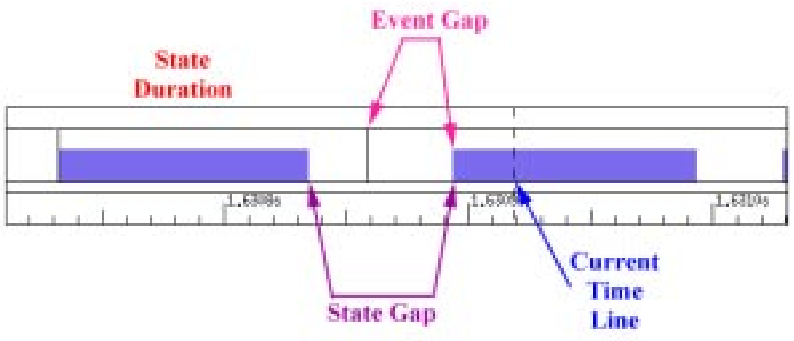


Figure 9-5. Function Terminology Illustrated

A more detailed example is illustrated in the following figure.

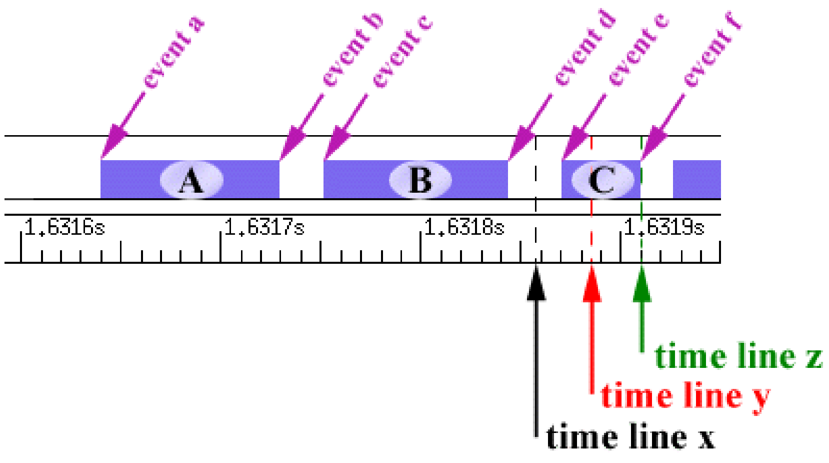


Figure 9-6. States and Events

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 9-6, 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 9-6.

`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 9-6, 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 9-6.

`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 9-6, 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 9-6.

`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 <u>end</u> 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 *qualified event* specified to that function or the *current trace event*. They include the following:

- `id()`
- `arg()`
- `arg_dbl()`
- `num_args()`
- `pid()`
- `raw_pid()`
- `lwpid()`
- `cpu()`
- `thread_id()`
- `task_id()`
- `tid()`
- `offset()`
- `time()`
- `node_id()`
- `pid_table_name()`
- `tid_table_name()`
- `node_name()`
- `process_name()`
- `task_name()`
- `thread_name()`
- Multi-event functions

id()

DESCRIPTION

The `id()` function returns the *trace event ID* of the last instance of a *trace event*.

SYNTAX

`id` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the *trace event ID* of the last instance of the trace event which satisfies the conditions of the specified qualified event. If omitted, the function returns the *trace event ID* of the current trace event. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

integer

SEE ALSO

“start_id()” on page 9-35, “end_id()” on page 9-46, and “offset_id()” on page 9-60.

arg()

DESCRIPTION

The `arg()` function returns the value of a particular *trace event argument*.

SYNTAX

`arg`[*N*] [([*QE*])]

PARAMETERS

N	Specifies the *N*th argument logged with the *trace event*. Defaults to 1.
QE	A user-defined *qualified event*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

integer

SEE ALSO

“arg_dbl()” on page 9-21, “num_args()” on page 9-21, “start_arg()” on page 9-35, “end_arg()” on page 9-47, and “offset_arg()” on page 9-60.

arg_dbl()

DESCRIPTION

The `arg_dbl()` function returns the value of a particular *trace event argument.*

SYNTAX

`arg`[*N*]`_dbl` [([*QE*])]

PARAMETERS

N	Specifies the *N*th argument logged with the *trace event*. Defaults to 1.
QE	A user-defined *qualified event*. If supplied, the function returns the specified argument for the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the specified argument for the *current trace event*. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

double-precision floating point

SEE ALSO

“arg()” on page 9-20, “num_args()” on page 9-21, “start_arg_dbl()” on page 9-36, “end_arg_dbl()” on page 9-47, and “offset_arg_dbl()” on page 9-61.

num_args()

DESCRIPTION

The `num_args()` function returns the number of arguments logged with a *trace event*.

SYNTAX

`num_args` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the number of arguments of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the number of arguments of the *current trace event*. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

integer

SEE ALSO

"arg()" on page 9-20, "start_num_args()" on page 9-37, "end_num_args()" on page 9-48, and "offset_num_args()" on page 9-61.

pid()

DESCRIPTION

The `pid()` function returns the global process identifier (*PID*) associated with a *trace event*.

NOTE

A global process identifier does not have the same meaning as the typical operating system definition of **`pid`**. A PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. Consult the **`_lwp_global_self(2)`** man page for more information.

SYNTAX

`pid` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the global process identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the global process identifier of the *current trace event*. For more information, see "Qualified Events" on page 9-81.

RETURN TYPE

integer

SEE ALSO

"PID List" on page 8-7, "raw_pid()" on page 9-23, "lwpid()" on page 9-23, "start_pid()" on page 9-37, "end_pid()" on page 9-48, and "offset_pid()" on page 9-62.

raw_pid()

DESCRIPTION

The `raw_pid()` function returns the process identifier (*raw PID*) associated with a *trace event*.

NOTE

A NightTrace raw PID has the same meaning as the typical operating system definition of **`pid`**. See the **`getpid(2)`** man page for more information.

SYNTAX

`raw_pid` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the process identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the process identifier of the *current trace event*. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

integer

SEE ALSO

“PID List” on page 8-7, “pid()” on page 9-22, “lwpid()” on page 9-23, “start_raw_pid()” on page 9-38, “end_raw_pid()” on page 9-49, and “offset_raw_pid()” on page 9-63.

lwpid()

DESCRIPTION

The `lwpid()` function returns the lightweight process identifier (*LWPID*) associated with a *trace event*.

NOTE

See the **`_lwp_self(2)`** man page for more information.

SYNTAX

`lwpid` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the lightweight process identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the lightweight process identifier of the *current trace event*. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

integer

SEE ALSO

“PID List” on page 8-7, “pid()” on page 9-22, “raw_pid()” on page 9-23, “start_lwpid()” on page 9-39, “end_lwpid()” on page 9-50, and “offset_lwpid()” on page 9-63.

thread_id()

DESCRIPTION

The `thread_id()` function returns the *thread* identifier associated with a *trace event*.

NOTE

See the **`thr_self(3thread)`** man page for more information.

SYNTAX

`thread_id` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the thread identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the thread identifier of the *current trace event*. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

integer

SEE ALSO

"start_thread_id()" on page 9-39, "end_thread_id()" on page 9-51, and "offset_thread_id()" on page 9-64.

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` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the Ada task identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the Ada task identifier of the *current trace event*. For more information, see "Qualified Events" on page 9-81.

RETURN TYPE

integer

SEE ALSO

"start_task_id()" on page 9-40, "end_task_id()" on page 9-51, and "offset_task_id()" on page 9-64.

tid()

DESCRIPTION

The `tid()` function returns the internally-assigned NightTrace thread identifier (*TID*) associated with a *trace event*.

SYNTAX

`tid` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the NightTrace thread identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the NightTrace thread identifier of the *current trace event*. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

integer

SEE ALSO

“TID List” on page 8-8, “start_tid()” on page 9-41, “end_tid()” on page 9-52, and “offset_tid()” on page 9-65.

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` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the logical CPU number of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the logical CPU number of the *current trace event*. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

integer

SEE ALSO

"start_cpu()" on page 9-41, "end_cpu()" on page 9-52, and "offset_cpu()" on page 9-66.

offset()

DESCRIPTION

The `offset()` function returns the ordinal number (*offset*) of a *trace event*.

SYNTAX

`offset` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the ordinal number (*offset*) of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the ordinal number (*offset*) of the *current trace event*. For more information, see "Qualified Events" on page 9-81.

RETURN TYPE

integer

SEE ALSO

"start_offset()" on page 9-42, "end_offset()" on page 9-53, "min_offset()" on page 9-73, and "max_offset()" on page 9-73.

time()

DESCRIPTION

The `time()` function returns the time, in seconds, associated with a *trace event*. Times are relative to the earliest trace event from all trace data files currently in use.

SYNTAX

`time` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the time, in seconds, of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the time, in seconds, of the *cur-*

rent trace event. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

double-precision floating point

SEE ALSO

“event_gap()” on page 9-32, “start_time()” on page 9-42, “end_time()” on page 9-54, “state_gap()” on page 9-57, “state_dur()” on page 9-57, and “offset_time()” on page 9-66.

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 9-30 for more information about this function.)

SYNTAX

`node_id` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the node identifier of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the node identifier of the *current trace event*. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

integer

SEE ALSO

“start_node_id()” on page 9-43, “offset_node_id()” on page 9-67, and “end_node_id()” on page 9-54.

pid_table_name()

DESCRIPTION

The `pid_table_name()` function returns the name of the internally-assigned NightTrace process identifier table (*PID table*) associated with a *trace event*.

SYNTAX

`pid_table_name` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the name of the process identifier table (*PID table*) of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the name of the process identifier table (*PID table*) of the *current trace event*. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

string

SEE ALSO

“start_pid_table_name()” on page 9-44, “offset_pid_table_name()” on page 9-67, and “end_pid_table_name()” on page 9-55

tid_table_name()

DESCRIPTION

The `tid_table_name()` function returns the name of the internally-assigned NightTrace thread identifier table (*TID table*) associated with a *trace event*.

SYNTAX

`tid_table_name` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the name of the thread identifier table (*TID table*) of the last

instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the name of the thread identifier table (*TID table*) of the *current trace event*. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

string

SEE ALSO

“start_tid_table_name()” on page 9-44, “offset_tid_table_name()” on page 9-68, and “end_tid_table_name()” on page 9-55

node_name()

DESCRIPTION

The `node_name()` function returns the name of the system from which a *trace event* was logged.

SYNTAX

`node_name` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the name of system from which the last instance of the trace event which satisfies the conditions for the specified qualified event was logged. If omitted, the function returns the name of the system from which the *current trace event* was logged. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

string

SEE ALSO

“start_node_name()” on page 9-45, “offset_node_name()” on page 9-68, and “end_node_name()” on page 9-56

process_name()

DESCRIPTION

The `process_name()` function returns the name of the process (*PID*) associated with a *trace event*.

SYNTAX

`process_name` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the name associated with the *PID* of the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the name associated with the *PID* of the *current trace event*. For more information, see "Qualified Events" on page 9-81.

RETURN TYPE

string

SEE ALSO

"offset_process_name()" on page 9-69

task_name()

DESCRIPTION

The `task_name()` function returns the name of the task associated with a *trace event*.

NOTE

This function is only meaningful for trace events which were logged from Ada tasking programs.

SYNTAX

`task_name` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the name of the task associated with the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the name of the task associated with the *current trace event*. For more information, see "Qualified Events" on page 9-81.

RETURN TYPE

string

SEE ALSO

"offset_task_name()" on page 9-69

thread_name()

DESCRIPTION

The `thread_name()` function returns the thread name associated with a *trace event*.

SYNTAX

`thread_name` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function returns the thread name associated with the last instance of the trace event which satisfies the conditions for the specified qualified event. If omitted, the function returns the thread name associated with the *current trace event*. For more information, see "Qualified Events" on page 9-81.

RETURN TYPE

string

SEE ALSO

"offset_thread_name()" on page 9-70

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 preceeding occurrence.

SYNTAX

`event_gap` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event*. If supplied, the function calculates the gap between the two most recent occurrences of events which satisfy the conditions of the specified qualilfied event. If omitted, the function calculates the gap between the current trace event and the event immediately preceeding it. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

double-precision floating point

SEE ALSO

“time()” on page 9-27, “state_gap()” on page 9-57, and “state_dur()” on page 9-57.

event_matches()

DESCRIPTION

The `event_matches()` function returns the number of occurrences of a *trace event* on or before the *current time line.*

SYNTAX

`event_matches` [([*QE*])]

PARAMETERS

QE	A user-defined *qualified event.* If supplied, the function calculates the number of occurrences of events which satisfy the conditions of the specified qualified event on or before the current time line. If omitted, the function calculates the number of occurrences of all events on or before the current time line. For more information, see “Qualified Events” on page 9-81.

RETURN TYPE

integer

SEE ALSO

“summary_matches()” on page 9-74.

State Functions

In its simplest form, a *state* is a region of source code 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
- End functions
- Multi-state functions

Start Functions

The start functions provide information about the start *event* of the *most recent instance of a state*. The state to which the start function applies is either the *qualified state* specified to the function, or the state being currently defined. Thus, if a qualfied state is not specified, start functions are only meaningful when used in *expressions* associated within a state definition. In addition, start functions should not be used in a recursive manner in a Start Expression; a start function should not be specified in a Start Expression that applies to the state definition containing that Start Expression. Conversely, an End Expression may include start functions that apply to the state definition containing that End Expression.

NOTE

Start functions provide information about the *most recent instance of a state*, whereas end functions (see “End Functions” on page 9-45) provide information about the *last completed instance of a state*.

Start functions include the following:

- `start_id()`
- `start_arg()`
- `start_arg_dbl()`
- `start_num_args()`
- `start_pid()`
- `start_raw_pid()`
- `start_thread_id()`
- `start_task_id()`
- `start_tid()`
- `start_lwpid()`
- `start_cpu()`

- `start_offset()`
- `start_time()`
- `start_node_id()`
- `start_pid_table_name()`
- `start_tid_table_name()`
- `start_node_name()`

start_id()

DESCRIPTION

The `start_id()` function returns the *trace event ID* of the start *event* of the *most recent instance of a state*.

SYNTAX

`start_id` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“id()” on page 9-19, “end_id()” on page 9-46, and “offset_id()” on page 9-60.

start_arg()

DESCRIPTION

The `start_arg()` function returns the value of a particular *trace event argument* associated with the start *event* of the *most recent instance of a state*.

SYNTAX

`start_arg`[*N*] [([*QS*])]

PARAMETERS

N	Specifies the *N*th argument logged with the start *event*. Defaults to 1.
QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“arg()” on page 9-20, “start_arg_dbl()” on page 9-36, “start_num_args()” on page 9-37, “end_arg()” on page 9-47, and “offset_arg()” on page 9-60.

start_arg_dbl()

DESCRIPTION

The `start_arg_dbl()` function returns the value of a particular *trace event argument* associated with the start *event* of the *most recent instance of a state.*

SYNTAX

`start_arg`[*N*]`_dbl` [([*QS*])]

PARAMETERS

N	Specifies the *N*th argument logged with the start *event*. Defaults to 1.
QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

double-precision floating point

SEE ALSO

“arg_dbl()” on page 9-21, “start_arg()” on page 9-35, “start_num_args()” on page 9-37, “end_arg_dbl()” on page 9-47, and “offset_arg_dbl()” on page 9-61.

start_num_args()

DESCRIPTION

The `start_num_args()` function returns the number of arguments associated with the start *event* of the *most recent instance of a state*.

SYNTAX

`start_num_args` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“start_arg()” on page 9-35, “num_args()” on page 9-21, “end_num_args()” on page 9-48, and “offset_num_args()” on page 9-61.

start_pid()

DESCRIPTION

The `start_pid()` function returns the global process identifier (*PID*) associated with the start *event* of the *most recent instance of a state*.

NOTE

A global process identifier does not have the same meaning as the typical operating system definition of **`pid`**. A PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. Consult the **`_lwp_global_self(2)`** man page for more information.

SYNTAX

`start_pid` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“pid()” on page 9-22, “raw_pid()” on page 9-23, “lwpid()” on page 9-23, “end_pid()” on page 9-48, and “offset_pid()” on page 9-62.

start_raw_pid()

DESCRIPTION

The `start_raw_pid()` function returns the process identifier (*raw PID*) associated with the start *event* of the *most recent instance of a state*.

NOTE

A NightTrace raw PID has the same meaning as the typical operating system definition of **`pid`**. See the **`getpid(2)`** man page for more information.

SYNTAX

`start_raw_pid` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“pid()” on page 9-22, “raw_pid()” on page 9-23, “lwpid()” on page 9-23, “end_pid()” on page 9-48, and “offset_pid()” on page 9-62.

start_lwpid()

DESCRIPTION

The `start_lwpid()` function returns the lightweight process identifier (*LWPID*) associated with the start *event* of the *most recent instance of a state*.

NOTE

See the **`_lwp_self(2)`** man page for more information.

SYNTAX

`start_lwpid` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“pid()” on page 9-22, “raw_pid()” on page 9-23, “lwpid()” on page 9-23, “end_pid()” on page 9-48, and “offset_pid()” on page 9-62.

start_thread_id()

DESCRIPTION

The `start_thread_id()` function returns the *thread* identifier associated with the start *event* of the *most recent instance of a state*.

NOTE

See the **`thr_self(3thread)`** man page for more information.

SYNTAX

`start_thread_id` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“thread_id()” on page 9-24, “end_thread_id()” on page 9-51, and “offset_thread_id()” on page 9-64.

start_task_id()

DESCRIPTION

The `start_task_id()` function returns the Ada task identifier associated with the start *event* of the *most recent instance of a state*.

NOTE

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

SYNTAX

`start_task_id` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“task_id()” on page 9-25, “end_task_id()” on page 9-51, and “offset_task_id()” on page 9-64.

start_tid()

DESCRIPTION

The `start_tid()` function returns the internally-assigned NightTrace thread identifier (*TID*) associated with the start *event* of the *most recent instance of a state*.

SYNTAX

`start_tid` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

integer

SEE ALSO

"tid()" on page 9-25, "end_tid()" on page 9-52, and "offset_tid()" on page 9-65.

start_cpu()

DESCRIPTION

The `start_cpu()` function returns the logical CPU number associated with the start *event* of the *most recent instance of a state*. CPUs are logically numbered starting at 0 and monotonically increase thereafter.

NOTE

This function is only valid when applied to events from Night-Trace kernel trace event files.

SYNTAX

`start_cpu` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that

state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“cpu()” on page 9-26, “end_cpu()” on page 9-52, and “offset_cpu()” on page 9-66.

start_offset()

DESCRIPTION

The `start_offset()` function returns the ordinal number (*offset*) of the start *event* of the *most recent instance of a state*.

SYNTAX

`start_offset` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“offset()” on page 9-27 and “end_offset()” on page 9-53.

start_time()

DESCRIPTION

The `start_time()` function returns the time, in seconds, associated with the start *event* of the *most recent instance of a state*. Times are relative to the earliest trace event from all trace data files currently in use.

SYNTAX

`start_time` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

double-precision floating point

SEE ALSO

“time()” on page 9-27, “end_time()” on page 9-54, “state_gap()” on page 9-57, “state_dur()” on page 9-57, and “offset_time()” on page 9-66.

start_node_id()

DESCRIPTION

The `start_node_id()` function returns the internally-assigned *node identifier* associated with the start *event* of the *most recent instance of a state*.

SYNTAX

`start_node_id` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“node_id()” on page 9-28, “offset_node_id()” on page 9-67, and “end_node_id()” on page 9-54

start_pid_table_name()

DESCRIPTION

The `start_pid_table_name()` function returns the name of the internally-assigned NightTrace process identifier table (*PID table*) associated with the start *event* of the *most recent instance of a state.*

SYNTAX

`start_pid_table_name` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

string

SEE ALSO

"pid_table_name()" on page 9-29, "offset_pid_table_name()" on page 9-67, and "end_pid_table_name()" on page 9-55

start_tid_table_name()

DESCRIPTION

The `start_tid_table_name()` function returns the name of the internally-assigned NightTrace thread identifier table (*TID table*) associated with the start *event* of the *most recent instance of a state.*

SYNTAX

`start_tid_table_name` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

string

SEE ALSO

“tid_table_name()” on page 9-29, “offset_tid_table_name()” on page 9-68, and “end_tid_table_name()” on page 9-55

start_node_name()

DESCRIPTION

The `start_node_name()` function returns the name of the system from which the start *event* of the *most recent instance of a state* was logged.

SYNTAX

`start_node_name` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

string

SEE ALSO

“node_name()” on page 9-30, “offset_node_name()” on page 9-68, and “end_node_name()” on page 9-56

End Functions

The end functions provide information about the end *event* of the *last completed instance of a state*. The *state* to which the end function applies is either the *qualified state* specified to the function, or the state being currently defined. Thus, if a qualfied state is not specified, end functions are only meaningful when used in expressions associated within a state definition.

NOTE

End functions provide information about the *last completed instance of a state*, whereas start functions (see “Start Functions” on page 9-34) provide information about the *most recent instance of a state*.

End functions include:

- `end_id()`
- `end_arg()`
- `end_arg_dbl()`
- `end_num_args()`
- `end_pid()`
- `end_raw_pid()`
- `end_lwpid()`
- `end_thread_id()`
- `end_task_id()`
- `end_tid()`
- `end_cpu()`
- `end_offset()`
- `end_time()`
- `end_node_id()`
- `end_pid_table_name()`
- `end_tid_table_name()`
- `end_node_name()`

end_id()

DESCRIPTION

The `end_id()` function returns the *trace event ID* associated with the end *event* of the *last completed instance of a state.*

SYNTAX

`end_id` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

"id()" on page 9-19, "start_id()" on page 9-35, and "offset_id()" on page 9-60.

end_arg()

DESCRIPTION

The `end_arg()` function returns the value of a particular *trace event argument* associated with the <u>end</u> *event* of the *last completed instance of a state*.

SYNTAX

`end_arg`[*N*] [([*QS*])]

PARAMETERS

N	Specifies the *N*th argument logged with the trace event. Defaults to 1.
QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

integer

SEE ALSO

"arg()" on page 9-20, "start_arg()" on page 9-35, "end_arg_dbl()" on page 9-47, "end_num_args()" on page 9-48, and "offset_arg()" on page 9-60.

end_arg_dbl()

DESCRIPTION

The `end_arg_dbl()` function returns the value of a particular *trace event argument* associated with the <u>end</u> *event* of the *last completed instance of a state*.

SYNTAX

`end_arg`[*N*]`_dbl` [([*QS*])]

PARAMETERS

N	Specifies the *N*th argument logged with the trace event. Defaults to 1.

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

double-precision floating point

SEE ALSO

“arg_dbl()” on page 9-21, “start_arg_dbl()” on page 9-36, “end_arg()” on page 9-47, “end_num_args()” on page 9-48, and “offset_arg_dbl()” on page 9-61.

end_num_args()

DESCRIPTION

The `end_num_args()` function returns the number of arguments associated with the end *event* of the *last completed instance of a state*.

SYNTAX

`end_num_args` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“num_args()” on page 9-21, “start_num_args()” on page 9-37, “end_arg()” on page 9-47, and “offset_num_args()” on page 9-61.

end_pid()

DESCRIPTION

The `end_pid()` function returns the global process identifier (*PID*) associated with the end *event* of the *last completed instance of a state*.

NOTE

> A global process identifier does not have the same meaning as the typical operating system definition of **`pid`**. A PID within NightTrace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the lightweight process identifier (*LWPID*) in the lower 16 bits. Consult the **`_lwp_global_self(2)`** man page for more information.

SYNTAX

`end_pid` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

integer

SEE ALSO

"pid()" on page 9-22, "raw_pid()" on page 9-23, "lwpid()" on page 9-23, "start_pid()" on page 9-37, and "offset_pid()" on page 9-62.

end_raw_pid()

DESCRIPTION

The `end_raw_pid()` function returns the process identifier (*raw PID*) associated with the <u>end</u> *event* of the *last completed instance of a state*.

NOTE

> A NightTrace raw PID has the same meaning as the typical operating system definition of **`pid`**. See the **`getpid(2)`** man page for more information.

SYNTAX

`end_raw_pid` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“pid()” on page 9-22, “raw_pid()” on page 9-23, “lwpid()” on page 9-23, “start_pid()” on page 9-37, and “offset_pid()” on page 9-62.

end_lwpid()

DESCRIPTION

The `end_lwpid()` function returns the lightweight process identifier (*LWPID*) associated with the end *event* of the *last completed instance of a state*.

NOTE

See the **`_lwp_self(2)`** man page for more information.

SYNTAX

`end_lwpid` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“pid()” on page 9-22, “raw_pid()” on page 9-23, “lwpid()” on page 9-23, “start_pid()” on page 9-37, and “offset_pid()” on page 9-62.

end_thread_id()

DESCRIPTION

The `end_thread_id()` function returns the *thread* identifier associated with the end *event* of the *last completed instance of a state.*

NOTE

See the **`thr_self(3thread)`** man page for more information.

SYNTAX

`end_thread_id` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“thread_id()” on page 9-24, “start_thread_id()” on page 9-39, and “offset_thread_id()” on page 9-64.

end_task_id()

DESCRIPTION

The `end_task_id()` function returns the Ada task identifier associated with the end *event* of the *last completed instance of a state.*

NOTE

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

SYNTAX

`end_task_id` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“task_id()” on page 9-25, “start_task_id()” on page 9-40, and “offset_task_id()” on page 9-64.

end_tid()

DESCRIPTION

The `end_tid()` function returns the internally-assigned NightTrace thread identifier (*TID*) associated with the end *event* of the *last completed instance of a state*.

SYNTAX

`end_tid` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

integer

SEE ALSO

“tid()” on page 9-25, “start_tid()” on page 9-41, and “offset_tid()” on page 9-65.

end_cpu()

DESCRIPTION

The `end_cpu()` function returns the logical CPU number associated with the end *event* of the *last completed instance of a state*. CPUs are logically numbered starting at 0 and monotonically increase thereafter.

NOTE

This function is only valid when applied to events from Night-Trace kernel trace event files.

SYNTAX

`end_cpu` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

integer

SEE ALSO

"cpu()" on page 9-26, "start_cpu()" on page 9-41, and "offset_cpu()" on page 9-66.

end_offset()

DESCRIPTION

The `end_offset()` function returns the ordinal number (*offset*) of the end *event* of the *last completed instance of a state*.

SYNTAX

`end_offset` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

integer

SEE ALSO

"offset()" on page 9-27 and "start_offset()" on page 9-42.

end_time()

DESCRIPTION

The `end_time()` function returns the time, in seconds, associated with the <u>end</u> *event* of the *last completed instance of a state*. Times are relative to the earliest trace event from all trace data files currently in use.

SYNTAX

`end_time` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

double-precision floating point

SEE ALSO

"time()" on page 9-27, "start_time()" on page 9-42, "state_gap()" on page 9-57, "state_dur()" on page 9-57, and "offset_time()" on page 9-66.

end_node_id()

DESCRIPTION

The `end_node_id()` function returns the internally-assigned *node identifier* associated with the <u>end</u> *event* of the *last completed instance of a state*.

SYNTAX

`end_node_id` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

integer

SEE ALSO

“node_id()” on page 9-28, “start_node_id()” on page 9-43, and “offset_node_id()” on page 9-67

end_pid_table_name()

DESCRIPTION

The `end_pid_table_name()` function returns the name of the internally-assigned NightTrace process identifier table (*PID table*) associated with the end *event* of the *last completed instance of a state*.

SYNTAX

`end_pid_table_name` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

string

SEE ALSO

“pid_table_name()” on page 9-29, “start_pid_table_name()” on page 9-44, and “offset_pid_table_name()” on page 9-67.

end_tid_table_name()

DESCRIPTION

The `end_tid_table_name()` function returns the name of the internally-assigned NightTrace thread identifier table (*TID table*) associated with the end *event* of the *last completed instance of a state*.

SYNTAX

`end_tid_table_name` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

string

SEE ALSO

"tid_table_name()" on page 9-29, "start_tid_table_name()" on page 9-44, and "offset_tid_table_name()" on page 9-68.

end_node_name()

DESCRIPTION

The `end_node_name()` function returns the name of the system from which the end *event* of the *last completed instance of a state* was logged.

SYNTAX

`end_node_name` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state.* If supplied, it specifies the state to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

string

SEE ALSO

"node_name()" on page 9-30, "start_node_name()" on page 9-45, and "offset_node_name()" on page 9-68.

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 “StateGraph” on page 8-14.

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 preceeding it or zero if there was no previous instance.

SYNTAX

`state_gap` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the *state* to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

double-precision floating point

SEE ALSO

“start_time()” on page 9-42, “end_time()” on page 9-54, “event_gap()” on page 9-32, and “state_dur()” on page 9-57.

state_dur()

DESCRIPTION

The `state_dur()` function returns the time in seconds between the start *event* and the end *event* of the *last completed instance of a state*. Thus, if the *current time line* occurs within an instance of the state but before it has ended, `state_dur()` returns the duration of the previous instance or zero if there was no previous instance.

SYNTAX

`state_dur` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the *state* to which the function applies. If omitted, the function may

	only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

double-precision floating point

SEE ALSO

"state_gap()" on page 9-57.

state_matches()

DESCRIPTION

The `state_matches()` function returns the number of completed instances of a state on or before the *current time line*.

SYNTAX

`state_matches` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the *state* to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see "Qualified States" on page 9-83.

RETURN TYPE

integer

SEE ALSO

"Start Functions" on page 9-34 and "summary_matches()" on page 9-74.

state_status()

DESCRIPTION

The `state_status()` function indicates whether the *current time line* resides within a *current instance of a state*. Thus, if the current time line is positioned in the region from the start *event* up to, but not including, the end *event* of an instance of the state, the return value is TRUE. Otherwise, it is FALSE.

SYNTAX

`state_status` [([*QS*])]

PARAMETERS

QS	A user-defined *qualified state*. If supplied, it specifies the *state* to which the function applies. If omitted, the function may only be used within a state definition and then applies to that state. For more information, see “Qualified States” on page 9-83.

RETURN TYPE

boolean

Offset Functions

All offset functions take an expression that evaluates to an ordinal trace event (*offset*) as a parameter. (Offsets begin at zero.) These functions include the following:

- `offset_id()`
- `offset_arg()`
- `offset_arg_dbl()`
- `offset_num_args()`
- `offset_pid()`
- `offset_raw_pid()`
- `offset_lwpid()`
- `offset_thread_id()`
- `offset_task_id()`
- `offset_tid()`
- `offset_cpu ()`
- `offset_time()`
- `offset_node_id()`
- `offset_pid_table_name()`
- `offset_tid_table_name()`
- `offset_node_name()`
- `offset_process_name()`
- `offset_task_name()`
- `offset_thread_name()`

Usually, these functions take one of the following functions as a parameter:

- `offset()`

- `start_offset()`
- `end_offset()`
- `min_offset()`
- `max_offset()`

For information about these functions, see “offset()” on page 9-27, “start_offset()” on page 9-42, “end_offset()” on page 9-53, “min_offset()” on page 9-73, and “max_offset()” on page 9-73.

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 9-19, “start_id()” on page 9-35, and “end_id()” on page 9-46.

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 9-20, "start_arg()" on page 9-35, "end_arg()" on page 9-47, "offset_arg_dbl()" on page 9-61, and "offset_num_args()" on page 9-61.

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 9-21, "start_arg_dbl()" on page 9-36, "end_arg_dbl()" on page 9-47, "offset_arg()" on page 9-60, and "offset_num_args()" on page 9-61.

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 9-21, “start_num_args()” on page 9-37, “end_num_args()” on page 9-48, “offset_arg()” on page 9-60, and “offset_arg_dbl()” on page 9-61.

offset_pid()

DESCRIPTION

The `offset_pid()` function returns the global process identifier (*PID*) from which the ordinal trace event (*offset*) was logged.

NOTE

A global process identifier does not have the same meaning as the typical operating system definition of **`pid`**. A PID within Night-Trace is a 32-bit integer value that contains the operating system process identifier (*raw PID*) in the upper 16 bits and the light-weight process identifier (*LWPID*) in the lower 16 bits. Consult the **`_lwp_global_self(2)`** man page for more information.

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 9-22, “raw_pid()” on page 9-23, “lwpid()” on page 9-23, “start_pid()” on page 9-37, and “end_pid()” on page 9-48.

offset_raw_pid()

DESCRIPTION

The `offset_raw_pid()` function returns the process identifier (*raw PID*) from which the ordinal trace event (*offset*) was logged.

NOTE

A NightTrace raw PID has the same meaning as the typical operating system definition of **`pid`**. See the **`getpid(2)`** man page for more information.

SYNTAX

`offset_raw_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 9-22, “raw_pid()” on page 9-23, “lwpid()” on page 9-23, “start_pid()” on page 9-37, and “end_pid()” on page 9-48.

offset_lwpid()

DESCRIPTION

The `offset_lwpid()` function returns the lightweight process identifier (*LWPID*) from which the ordinal trace event (*offset*) was logged.

NOTE

See the **`_lwp_self(2)`** man page for more information.

SYNTAX

`offset_lwpid` (*offset_expr*)

PARAMETERS

offset_expr	An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

integer

SEE ALSO

“pid()” on page 9-22, “raw_pid()” on page 9-23, “lwpid()” on page 9-23, “start_lwpid()” on page 9-39, and “end_lwpid()” on page 9-50.

offset_thread_id()

DESCRIPTION

The `offset_thread_id()` function returns the *thread* identifier from which the ordinal trace event (*offset*) was logged.

NOTE

See the **`thr_self(3thread)`** man page for more information.

SYNTAX

`offset_thread_id` (*offset_expr*)

PARAMETERS

offset_expr	An expression that evaluates to the *offset* (or ordinal trace event number) of a trace event.

RETURN TYPE

integer

SEE ALSO

“thread_id()” on page 9-24, “start_thread_id()” on page 9-39, and “end_thread_id()” on page 9-51.

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 9-25, “start_task_id()” on page 9-40, and “end_task_id()” on page 9-51.

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 9-25, “start_tid()” on page 9-41, and “end_tid()” on page 9-52.

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 9-26, “start_cpu()” on page 9-41, and “end_cpu()” on page 9-52.

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 9-27, "start_time()" on page 9-42, and "end_time()" on page 9-54.

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 9-28, "start_node_id()" on page 9-43, and "end_node_id()" on page 9-54

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 9-29, "start_pid_table_name()" on page 9-44, and "end_pid_table_name()" on page 9-55

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 9-29, "start_tid_table_name()" on page 9-44, and "end_tid_table_name()" on page 9-55

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 9-30, "start_node_name()" on page 9-45, and "end_node_name()" on page 9-56

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 9-30

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 9-31

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 9-32

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 9-70 and “Summarizing Statistical Information” on page 10-5.

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 9-70 and "Summarizing Statistical Information" on page 10-5.

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 9-70 and "Summarizing Statistical Information" on page 10-5.

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 9-70 and "Summarizing Statistical Information" on page 10-5.

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 9-70 and "Summarizing Statistical Information" on page 10-5.

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 9-70 and "Summarizing Statistical Information" on page 10-5.

summary_matches()

DESCRIPTION

The `summary_matches()` function returns the number of times the summary criteria <u>and</u> `Filter-Expression` were matched in the time range.

NOTE

This function should only used in the `Summarize Form`. Its behavior elsewhere is undefined. (See "Summarizing Statistical Information" on page 10-5 for more information.)

SYNTAX

`summary_matches` ()

RETURN TYPE

integer

SEE ALSO

"event_matches()" on page 9-33 and "state_matches()" on page 9-58. For information about `Filter-Expression`, see "Summarize Form Fields" on page 10-6.

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 "ntrace Tables" on page 5-13 and "Kernel String Tables" on page 11-32.

get_string()

The `get_string()` routine dynamically looks up a string in a string table.

SYNTAX

`get_string` (*table_name*[, *int_expr*])

PARAMETERS

table_name — *table_name* is an unquoted character string that represents the name of a string table. To avoid possible forward reference problems, try to make your `get_string()` calls refer to previously-defined string tables. The following string table names are pre-defined in NightTrace: `event`, `pid`, `tid`, `boolean`, `name_pid`, `name_tid`, `node_name`, `pid_`*nodename*, `tid_`*nodename*, `vector`, `syscall`, `device`, `event_summary`, `event_arg_summary`, `event_arg_dbl_summary`, `state_summary`. For more information on these tables, see "Pre-Defined String Tables" on page 5-15 and "Kernel String Tables" on page 11-32.

int_expr — *int_expr* is an integer expression that acts as an index into the specified string table. *int_expr* must either match an identifying integer value in the *table_name* string table, or the *table_name* string table must have a default item line; otherwise `get_string()` returns a string of *int_expr* in decimal. Often *int_expr* is based on a NightTrace function.

If your table consists of only a default item line, omit this parameter.

DESCRIPTION

The following NightTrace constructs can call `get_string()` to dynamically locate a static string in a string table:

- A `Then-Expression` of a display object configuration
- A value field of a format table

For each `get_string()` call, NightTrace follows these steps:

1. Evaluates *int_expr*
2. Uses this value as an index into *table_name*
3. Retrieves the associated string from *table_name*
4. Returns a string

The following lines provide a brief example of a call to `get_string()`.

```
string_table (conditions) = {
   item =  1, “normal”;
   item = 50, “YELLOW ALERT”;
   item = 99, “RED ALERT”;
   default_item = “N/A”;
};
```

In this example the numeric argument associated with a trace event represents the current conditions (`conditions`). If the argument has the value 99, NightTrace:

1. Uses the value 99 as in index into `conditions`
2. Retrieves the associated string (“`RED ALERT`”) from `conditions`
3. Returns “`RED ALERT`”

RETURN TYPES

On successful completion, `get_string()` returns a string from a string table. NightTrace returns a string of the item number, *int_expr*, in decimal if *table_name* is not found, or if *int_expr* is not found and there is no default item line. The first time *table_name* is not found, NightTrace issues an error message. Because `get_string()` returns a string, you can use it anywhere a string expression is appropriate.

For more information on string tables, see “String Tables” on page 5-14, Table 8-3, and the **`/usr/lib/NightTrace/tables`** file.

get_item()

The `get_item()` routine looks up an item number in a string table.

SYNTAX

`int get_item` (*table_name*, “*str_const*”)

PARAMETERS

table_name	*table_name* is an unquoted character string that represents the name of a string table. To avoid possible forward reference problems, try to make your `get_item()` calls refer to previously-defined string tables. The following string table names are pre-defined in NightTrace: `event, pid, tid, boolean, name_pid, name_tid, node_name, pid_`*nodename*`, tid_`*nodename*`, vector, syscall, device, event_summary, event_arg_summary, event_arg_dbl_summary, state_summary`. For more information on these tables, see “Pre-Defined String Tables” on page 5-15 and “Kernel String Tables” on page 11-32.
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

Usually you would put a `get_item()` call in a `Then-Expression` of a display object configuration to locate an index number in a string table. For each `get_item()` call, NightTrace follows these steps:

1. Uses *str_const* value as an index into *table_name*
2. Retrieves the first associated index number from *table_name*
3. Returns the index number

Assume that the following string table definition is in your configuration file.

```
string_table (fruit) = {
   item = 3, “apple”;
   item = 4, “orange”;
   item = 5, “cherry”;
   item = 6, “banana”;
   default_item = “Unknown”;
};
```

Assume that you make the following call in the `Then-Expression` of a DataBox.

```
get_item (fruit, “orange”)
```

In this example, the `fruit` string table associates specific numeric codes with a corresponding fruit name string; it associates all other numeric codes with the string `“Unknown.”` When NightTrace evaluates the `Then-Expression` of this DataBox, it:

1. Calls `get_item()`
2. Uses the string `“orange”` as an index into the `fruit` string table
3. Retrieves the (first) associated index (4)
4. Returns the index number (4)

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 5-14 and the **`/usr/lib/NightTrace/tables`** file. For more examples of function calls with pre-defined string tables, see Table 8-3.

get_format()

The `get_format()` routine dynamically looks up a string in a format table.

SYNTAX

`get_format` (*table_name*[, *int_expr*])

PARAMETERS

table_name	*table_name* is an unquoted character string that represents the name of a format table. To avoid possible forward reference problems, try to make your `get_format()` calls refer to previously-defined format tables.
int_expr	*int_expr* is an integer expression that acts as an index into the specified format table. *int_expr* must either match an identifying integer value in the *table_name* format table, or the *table_name* format table must have a default item line; otherwise, the results are undefined. Often *int_expr* is based on a NightTrace function. If your table consists of only a default item line, omit this parameter.

DESCRIPTION

A call to `get_format()` must be the first function call in an expression. You must not nest calls to `get_format()`.

The `Then-Expression` parameter of a DataBox configuration and the `Summarize-Expression` on a `Summary Form` can call `get_format()` to dynamically locate a string in a format table. For each `get_format()` call, NightTrace follows these steps:

1. Evaluates *int_expr*
2. Uses this value as an index into *table_name*
3. Retrieves the associated string from *table_name*
4. Replaces any conversion specifications in the associated string
5. Returns a string

Assume that the following format table definition is in your configuration file.

```
format_table (what_pid) = {
   item = 1, "Trace event 1 logged by pid %d'%d", "raw_pid()",
             "lwpid()";
   default_item = "Unaccounted for event ID (%d)", "id()";
};
```

Assume that you make the following call in the `Then-Expression` of a DataBox.

```
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 5-18 and the **`/usr/lib/NightTrace/tables`** file. For more examples of function calls with pre-defined format tables, see Table 8-3.

format()

The `format()` routine displays a string.

SYNTAX

`format` ("*format_string*" [, *arg*] ...)

PARAMETERS

format_string	*format_string* controls how the optional *args* are displayed. *format_string* is based on the format parameter used in the **`printf(3S)`** routine in C. It is a character string enclosed in double quotes that contains literal characters and conversion specifications. The literals are copied as is to the display object. Conversion specifications modify zero or more *args*.
arg	*arg* is an optional expression to be formatted and displayed.

DESCRIPTION

Call the `format()` function to display a string. You can do this only from the `Then-Expression` parameter of a display object configuration or the

`Summary-Expression` of the `Summarize Form`. A call to `format()` must be the first function call in an expression. You must not nest calls to `format()`.

The following lines provide examples of `format()` statements and what they display. Assume all variables have a value of 10 (decimal).

```
format( "Error”)                      Error
format( "Event=%d”, id() )            Event=10
format( "Argument is %X”, arg1())     Argument is A
```

RETURN TYPES

On successful completion, `format()` returns a string. Otherwise, it returns an empty string.

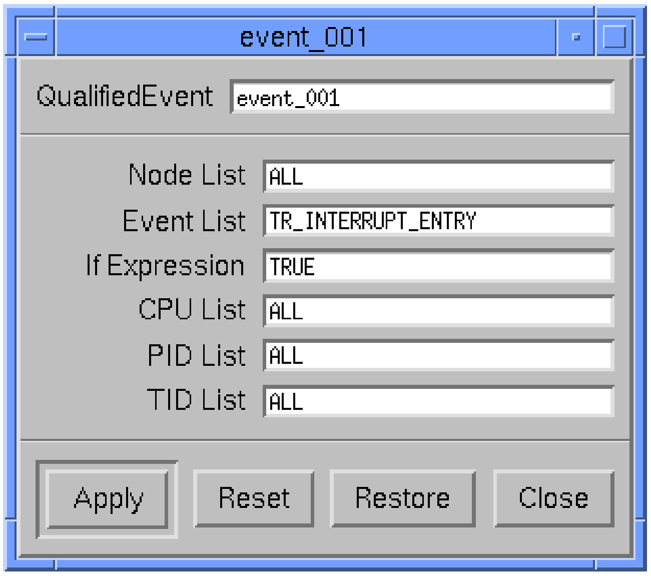
Qualified Events

A *qualified event* is a user-defined **named** *event* configuration that consists of a set of one or more trace events, possibly restricted by an `If-Expression`, `CPU List`, `TID List`, `PID List`, and `Node List`. Qualified events provide a mechanism for referencing *trace event configurations* within some *functions*; for example, they cannot appear alone in a DataBox configuration.

You may use a qualified event in trace event functions. For more information, see “Trace Event Functions” on page 9-19.

To create a qualified event definition, select the `Qualified Events` menu item from the `Expressions` menu (see “Expressions Menu” on page 9-1) to open the `Qualified Events Dialog Box` (see “Expression Dialog Boxes” on page 9-2 for details on this type of dialog).

Click the `Add` button on the `Qualified Events Dialog Box`, select the qualified event from the list, and click on the `Configure` button to pop up a `Qualified Event Configuration Form`, like the one shown in Figure 9-7.



QualifiedEvent	event_001
Node List	ALL
Event List	TR_INTERRUPT_ENTRY
If Expression	TRUE
CPU List	ALL
PID List	ALL
TID List	ALL

Figure 9-7. Qualified Event Configuration Form

The following parameter is specific to the Qualified Event Configuration Form.

QualifiedEvent	The name by which you refer to this qualified event in expressions.

TIP:

Consider giving your trace events upper case names in event-map files and giving any corresponding qualified event the same name in lower case.

NOTE

> The Node List field appears in this dialog only when NightTrace is configured to use an *RCIM* to timestamp events. (See “Node List” on page 8-9 for more information about this field.)

For information about other configuration parameters, see Chapter 8, especially “Common Configuration Parameters” on page 8-1.

Configuring qualified events is similar to configuring DataBox display objects. The configuration parameters for a qualified event are identical to those that are used to configure a DataBox display object. See “DataBox” on page 8-13 for information on how to configure a DataBox.

EXAMPLE

Qualified events can be useful when you are interested in seeing a trace event (or state) that occurs within a certain amount of time after another trace event. Given the following qualified event configuration:

```
QualifiedEvent: fire
Event List:     FIRE
CPU List:       2
```

an EventGraph can be configured to show only BAR trace events that happen within 100 microseconds of a FIRE trace event on CPU 2:

```
Event List:     BAR
If Expression: time() - time(fire) < 100us
```

Note: The BAR trace events themselves can happen on any CPU, and as long as they occur with 100 microseconds of a FIRE trace event on CPU 2, they will be graphed.

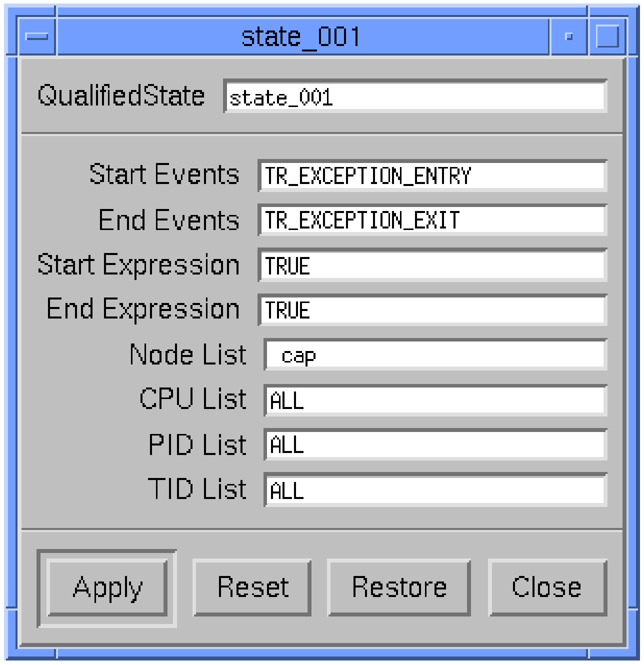
Qualified States

A *qualified state* is a user-defined **named** *state* configuration that consists of a set of one or more states, possibly restricted by a Start-Expression, End-Expression, CPU List, TID List, PID List, and Node List. Qualified states provide a mechanism for referencing state *configurations* within some *functions*.

You may use a qualified state in the following predefined functions: start functions, end functions, and multi-state functions. For more information, see “Start Functions” on page 9-34, “End Functions” on page 9-45, and “Multi-State Functions” on page 9-56.

To create a qualified state definition, select the Qualified States menu item from the Expressions menu (see “Expressions Menu” on page 9-1) to open the Qualified States Dialog Box (see “Expression Dialog Boxes” on page 9-2 for details on this type of dialog).

Click the Add button on the Qualified States Dialog Box, select the qualified state from the list, and click on the Configure button to pop up a Qualified State Configuration Form, like the one shown in Figure 9-8.



QualifiedState	state_001
Start Events	TR_EXCEPTION_ENTRY
End Events	TR_EXCEPTION_EXIT
Start Expression	TRUE
End Expression	TRUE
Node List	cap
CPU List	ALL
PID List	ALL
TID List	ALL

Figure 9-8. Qualified State Configuration Form

The following parameter is specific to the Qualified State Configuration Form.

QualifiedState	The name by which you refer to this qualified state in expressions.

NOTE

The Node List field appears in this dialog only when NightTrace is configured to use an *RCIM* to timestamp events. (See "Node List" on page 8-9 for more information about this field.)

For information about other configuration parameters, see Chapter 8, especially "Common Configuration Parameters" on page 8-1 and "StateGraph" on page 8-14.

Configuring qualified states is similar to configuring StateGraph display objects. The configuration parameters for a qualified state are identical to those that are used to configure a StateGraph display object. See "StateGraph" on page 8-14 for information on how to configure a StateGraph.

EXAMPLE

Qualified states can be useful when you are interested in a trace event that occurs while a certain state is active. The following qualified state:

```
    QualifiedState: foo_state
    Start Events:   PROG_A_BEGIN
    End Events:     PROG_A_EXIT
```

defines a state that is active whenever program `A` is running. Assume that another process is logging `FOO` trace events asynchronously. If you are interested only in the `FOO` trace events that are logged while program `A` is running, you can define an EventGraph as follows:

```
    Event List:     FOO
    If Expression: state_status(foo_state) == true
```

This graphs only `FOO` trace events that occur while the qualified state `foo_state` is active. (The “`== true`” is not necessary.) Thus, you see only `FOO` trace events logged while program `A` is running.

10
Using the Built-In Tools

10
Using the Built-In Tools

Overview

ntrace comes with a set of built-in tools available in View mode. These tools make it easier for you to pinpoint important trace events and numerically analyze aspects of your trace session.

This chapter covers the following built-in tools:

Search	Locates interesting parts of your trace session
Summarize	Summarizes statistics about trace events or states

Figure 10-1 shows the display page menu that gives you access to these tools.

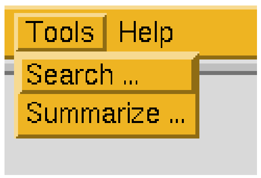


Figure 10-1. Tools Menu

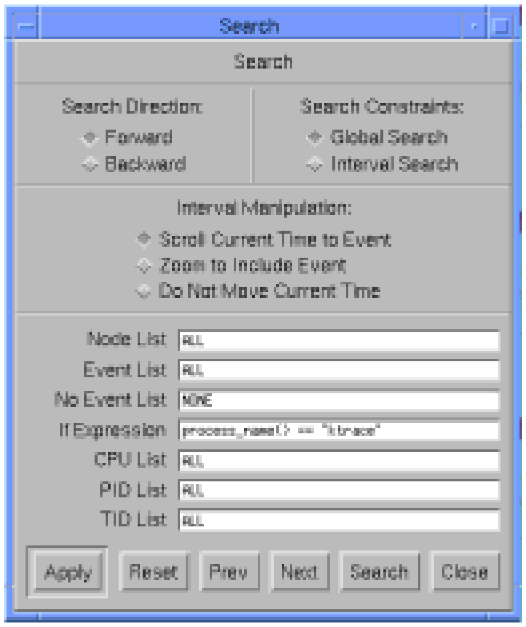
Searching for Points of Interest

Clicking on Tools ➪ Search … on the display page allows you to locate areas of interest in your trace event file(s). When you click on Tools ➪ Search …, the Search Form appears. This form lets you provide search specifications and define conditions you wish to find in your trace event file(s).

The Search Form consists of:

- Radio buttons
- Push buttons
- Text fields

Figure 10-2 illustrates the Search Form.



Search Direction: [✓ Forward] Backward

Search Constraints: [✓ Global Search] Interval Search

Interval Manipulation: [✓ Scroll Current Time to Event] Zoom to Include Event Do Not Move Current Time

Node List	ALL
Event List	ALL
No Event List	NONE
If Expression	process_name() == "ktrace"
CPU List	ALL
PID List	ALL
TID List	ALL

Figure 10-2. The Search Form

NOTE

The Node List field appears in this dialog only when NightTrace is configured to use an RCIM to timestamp events.

Search Form Radio Buttons

Through the Search Form's radio buttons, you can choose:

- The direction of a search
- The interval to search
- The effect of a search on the grid and interval control area of a display page

The Search Direction radio buttons let you search forward or backward in your trace session, relative to the current time.

- Click on the Forward radio button to search through newer trace events. This is the default setting.
- Click on the Backward radio button to search through older trace events. Note: This is a much less efficient search than a forward search.

The Search Constraints radio buttons let you limit your search to the entire trace session or to the current interval.

- Click on the Global Search radio button to search from the current time through the end (or beginning) of the trace session. This is the default setting.
- Click on Interval Search to search only between this interval's Time Start and Time End.

The Interval Manipulation radio buttons let you choose the action **ntrace** takes if a trace event meets all your criteria. This decision can affect both the grid and the interval control area.

- Click on Scroll Current Time to Event if you want **ntrace** to set the current time to the time when the trace event occurred and move the interval. This is the default setting.
- Click on Zoom to Include Event to zoom out the interval end time (for forward searches) or the interval start time (for backward searches) to include the found trace event. Clicking this radio button also updates the current time.
- Click on Do Not Move Current Time if you want **ntrace** to just write a message to the message display area of the display page without repositioning you on the grid or in the interval control area; a side-effect of this setting is that repeatedly clicking on the Search push button does not find trace events after the first one found. This is because the current time has not changed.

Search Form Push Buttons

Following is a summary of the effects of clicking on the push buttons in the Search Form:

Apply	(default) Validates any field change(s) on the Search Form. Clicking on Apply is equivalent to pressing <Enter>.
Reset	Restores changed field(s) on the Search Form to the value(s) they had after the last Apply or <Enter>. This works only if you have not already pressed <Enter> or clicked on the Apply push button. Clicking on Reset is equivalent to pressing <Esc>.
Prev	Goes backward one group of field settings in the search history and displays those settings in the fields. You may click on this push button multiple times to go backward several groups of settings. Clicking on this push button from the earliest group of settings has no effect. This push button is useful only after you have clicked on the Search push button.
Next	Goes forward one group of field settings in the search history and displays those settings in the fields. You may click on this push button multiple times to go forward several groups of settings. Clicking on this push button from the most recent group of settings has no effect. This push button is useful only after you have clicked on the Search and Prev push buttons.

Close — Closes the Search Form window and erases all but the last group of field settings from the search history. That is, if you click on Close and reopen this window during the same **ntrace** session, **ntrace** displays your most recent field settings; until you save more field settings, clicking on Prev and Next have no effect.

Search — Performs a search starting at the current time and saves your field changes, but not your radio button settings.

- Clicking on this push button causes **ntrace** to search through your trace event file(s) based on the criteria from the Search Form fields and the radio button settings.
- If you have made a field change, clicking on this push button makes **ntrace** temporarily save your field settings in the search history in memory. By saving your field settings in the search history, **ntrace** gives you an easy way to retrieve groups of field settings for use in future searches.

Because all fields and radio buttons on the Search Form have default settings, you can click on the Search push button without modifying anything in this window. The default search behavior is:

- Search forward through the entire trace session for any trace event from any process on any CPU.
- If a trace event meets all these criteria, **ntrace**:
  - Writes an informative message in the message display area of the display page that tells which ordinal trace event (offset) it found.
  - Sets the current time to the time when the trace event occurred.
  - Updates the grid and fields in the interval control area of the display page.
- If no trace event meets all these criteria, **ntrace** writes an error message in the message display area of the display page that tells from which ordinal trace event (offset) it began the search.

Search Form Fields

All fields of the Search Form have default values. Because of these defaults, clicking on Search without making any field changes makes **ntrace** search for the next (or previous) trace event in your trace event file(s). If you want to restrict this operation by trace event ID, trace event tag, CPU number, node, process name or global process identifier (PID), thread name or NightTrace thread identifier, or expression, you can do that by editing one or more of the fields on the Search Form. You can restore a field to its default value by entering a single space character or the word `default` into the field and clicking Apply or pressing <Enter>.

When you have finished editing the Search Form fields, press <Enter> or click on Apply. This causes **ntrace** to validate the data in each field you modified. For general information on field editing and how **ntrace** handles editing errors, see “Field Editing” on page 6-16.

When you are ready for **ntrace** to do a search, click on the Search push button. **ntrace** logical-ORs comma-separated lists of values within a field and logical-ANDs fields’ values. This means that a trace event must match at least one entry in each list and all criteria from the fields. If **ntrace** locates a trace event that meets every field criterion, it writes an informative message in the message display area on the display page. Depending on your preferences, it may also reposition the interval and current time line. If **ntrace** does not locate a trace event that meets every field criterion, it writes an error message in the message display area on the display page. For more information on the Search push button, see “Search Form Push Buttons” on page 10-3.

When you make field changes and click on Search, **ntrace** temporarily saves your field settings in the search history in memory. You can step through these groups of settings by clicking on the Prev and Next push buttons. Clicking on the Close push button erases all but the last group of field settings from the search history. For more information on these push buttons, see “Search Form Push Buttons” on page 10-3.

See Chapter 8 for a definition of each field, all its possible values, and its default value. There is only one difference between the **ntrace** behavior described there and the behavior of the Search Form: on the Search Form **ntrace** searches for, but does not display, data that meets the criteria. The search stops when **ntrace** finds a suitable value or runs out of trace events.

The No Event List field is the only field that is unique to the Search Form. This field lets you decide which trace event(s) to ignore in a search. The possible values are the same as those in the Event List field. It is not meaningful to put the same value in the Event List and in the No Event List.

NOTE

The Node List field appears in this dialog only when NightTrace is configured to use an RCIM to timestamp events.

Summarizing Statistical Information

Clicking on Tools ➪ Summarize … on the display page lets you get statistical information about trace events and states. When you click on Tools ➪ Summarize …, the Summarize Form appears. This form lets you constrain the information to be summarized.

The Summarize Form consists of:

- Radio buttons
- Text fields
- Summary display area

- Push buttons
- Menu items

Figure 10-4 and Figure 10-5 show two Summarize Forms with different configurations.

Summarize Form Radio Buttons

Through the Summarize Form's radio buttons, you can choose:

- Whether to summarize trace events or states
- The interval to summarize

The Summary Type radio buttons let you specify the type of information you want summarized.

- Click on the Event radio button to summarize trace event information. This is the default setting.
- Click on the State radio button to summarize state information.

The Summary Range radio buttons let you limit the summary to the current interval, to the time between a mark and the current time, or to the entire trace session.

- Click on the Trace Event File radio button to summarize data throughout the trace session. This is the default setting.
- Click on Region to summarize data only between the mark and the current time.
- Click on Interval to summarize data only between the current interval's Time Start and Time End.

Summarize Form Fields

All fields of the Summarize Form have default values. Because of these defaults, clicking on Summarize without making any field or radio button changes makes **ntrace** summarize all trace events in your trace event file(s). If you want to restrict the summary by trace event ID, trace event tag, CPU number, node, process name or global process identifier, thread name or NightTrace thread identifier, or expression, you can do that by editing one or more of the fields on the Summarize Form.

When you have finished editing the Summarize Form fields, press <Enter> or click on Apply. This causes **ntrace** to validate the data in each field you modified. For general information on field editing and how **ntrace** handles editing errors, see "Field Editing" on page 6-16.

When you are ready for **ntrace** to summarize data, click on the Summarize push button. **ntrace** logical-ORs comma-separated lists of values within a field and logical-ANDs fields' values. This means that a summary object must match at least one entry in each list and all criteria from the fields. Every time you click on Summarize, **ntrace** writes lines of statistics in the summary display area. For more information on the Summarize push button, see “Summarize Form Push Buttons” on page 10-8.

The text fields on the Summarize Form differ depending on the selected summary type. See Chapter 8 for a definition of each field (except those described below) and all its possible values. There is only one difference between the **ntrace** behavior described there and the behavior of the Summarize Form: on the Summarize Form, **ntrace** textually summarizes all data, rather than displaying individual values that meet the criteria.

The following text describes fields specific to the Summarize Form, their possible and default values, and how the Summarize push button behaves when you modify that field.

Filter-Expression	This text field has all the characteristics of If-Expression, except it is evaluated only if the If-Expression (for trace event summaries) or End-Expression (for state summaries) are true. Values may be: a boolean **ntrace** expression, the word TRUE, or the word FALSE. The default is TRUE. When you click on Summarize, **ntrace** evaluates the expression for every trace event it summarizes. A FALSE in this field essentially disables the summary.
Summary-Expression	This text field is evaluated every time the If-Expression or End-Expression and Filter-Expression configuration criteria for your summary are met. It lets you specify the format of the summary text. Values may be: a call to the `format()` or `get_format()` function. The default is a `get_format(event_summary)` call for trace event summaries. For state summaries, the default is a `get_format(state_summary)` call. For more information about these format tables, see “Pre-Defined Format Tables” on page 5-21.

For example, if you wanted to limit your summary to trace events with a first argument value between 5 and 100, your If-Expression would look something like:

```
arg1() > 5 && arg1() < 100
```

If you wanted to determine the largest of these argument values, your Summary-Expression would look something like:

```
max( arg1() )
```

In another example, the following configuration:

```
Event List:100
If Expression:TRUE
Summary Expression:min( arg1() )
```

prints out the minimum value of the first argument of every trace event logged with a trace event ID of 100. To find the offset where this minimum occurred, set:

```
Summary Expression:min_offset( arg1() )
```

If you want both statistics, use the following:

```
        Summary Expression:format( "min %d at %d",
         min(arg1()), min_offset(arg1()) )
```

TIP:

If you are interested in many statistics or if you are going to reuse this summary format at a later date, consider defining and using a format table. For example,

```
    Summary Expression:  get_format( my_table )
```

The lack of a second parameter indicates that the only entry in format table `my_table` is the default item line. The pre-defined `event_arg_summary` format table has four formats defined in it. Format 1 produces summary data on `arg1`, format 2 does the same for `arg2`, etc.

For more information about format tables, see "Format Tables" on page 5-18 and the end of **`/usr/lib/NightTrace/tables`**.

TIP:

The `min_offset()` and `max_offset()` functions return the offset of the first trace event where the expression minimum or maximum was seen. Thus, if the same minimum or maximum was seen more than once, the offset corresponds to the first one seen.

TIP:

Including `min_offset()`, `max_offset()`, `min()`, or `max()` in your summary text tells you the inclusive range of matches that you summarized, and the `summary_matches()` function tells the number of matches that you summarized.

TIP:

Sometimes there are anomalies in the trace information logged by an application, such as an unusually long state duration during program start up; this can throw off the duration statistics when analyzing "typical" program performance. You can use the Start-Expression for state summaries and the If-Expression for event summaries to limit the range of trace events summarized and remove extraneous trace events from the statistics produced.

See Chapter 8 for information about configuration parameters. See Chapter 9 for information on **`ntrace`** expressions. For more information on the Summarize push button, see "Summarize Form Push Buttons" on page 10-8.

Summarize Form Push Buttons

Following is a summary of the effects of clicking on the push buttons in the Summarize Form:

Apply	(default) Validates any field change(s) on the Summarize Form. Clicking on Apply is equivalent to pressing <Enter>.
Reset	Restores changed field(s) on the Summarize Form to the value(s) they had after the last Apply or <Enter>. This works only if you

have not already pressed <Enter> or clicked on the Apply push button. Clicking on Reset is equivalent to pressing <Esc>.

Restore	Restores changed field(s) on the Summarize Form to the original value(s) they had when you brought up the form.
Clear	Erases all text in the summary display area.
Summarize	Saves your field changes and summarizes the requested data.

- If you have made a field change, clicking on this push button makes **ntrace** temporarily save your field settings.
- Clicking on this push button causes **ntrace** to summarize summary data from your trace event file(s) based on the criteria from the Summarize Form fields and the radio button settings.

Because all fields and radio buttons on the Summarize Form have default settings, you can click on the Summarize push button without modifying anything in this window. The default summarize behavior is:

- **ntrace** writes statistical messages in the summary display area that tell about trace event data through the entire trace session for any trace event from any process on any CPU.

If you have configured the Summarize Form by specifying additional criteria, the summarize behavior is:

- If a trace event or state meets all these criteria, **ntrace** writes statistical messages in the summary display area that tell: the trace events or states involved, and minimum, maximum, average, and total for intervals and/or trace event arguments.
- If no trace event or state meets all these criteria, **ntrace** writes a message in the summary display area that says that there are no trace event or state matches to summarize.

Menu Bar

The menu bar of the Summarize Form consists of the following menu item:

- File

File Operations

When you click on the File menu item on the Summarize Form, the pull-down menu shown in Figure 10-3 appears.

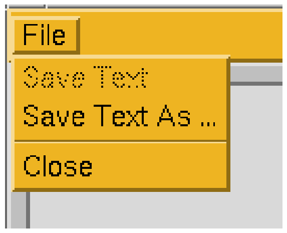


Figure 10-3. Summarize Form File Menu

Save Text

When you click on File ➪ Save Text on the Summarize Form, **ntrace** saves your summary text to the file you saved to last time. Any changes you have made since the last Save Text or Save Text As ... operation will be saved. You can continue running summaries after this operation. The Save Text operation is disabled (dimmed) if you have not both done a Save Text As ... and changed the summary display. Instead, use Save Text As ...

Save Text As ...

When you click on File ➪ Save Text As ... on the Summarize Form, **ntrace** saves your summary text to the specified file. You can continue running summaries after this operation.

Save Text As ... uses a File Selection Dialog Box to prompt you for a file name. See "The File Selection Dialog Box" on page 5-34 for more information.

Close

When you click on File ➪ Close on the Summarize Form, **ntrace** ends the current summary session, resets all field and radio button settings, and clears the summary display area. It does not prompt you to save your summary text since the last time you did a Save Text or Save Text As .... Therefore, if you have made any changes to the summary display area that you want to keep, you must perform a Save Text or Save Text As ... before you do a Close.

Summary Display Area

After you click on Summarize, **ntrace** appends statistics to the end of the scrolling summary display area. It automatically scrolls this area so the newest statistics are visible. Every line in this area has a unique number. A blank line separates sets of statistics. If you want new statistics to appear alone in the summary display area, click on Clear before you click on Summarize. See "Summarize Form Push Buttons" on page 10-8 for more information.

By default, **ntrace** displays 14 lines in the summary display area. You can alter this number by changing the size of the Summarize Form. To change the Summarize

Form size, vertically resize your window by using features of your window manager. It is not necessary to resize a window to see lines 15 and higher; you can scroll through all lines by using the scroll bar. Sometimes the statistical information exceeds the width of the summary display area. In this case, you must horizontally resize your window.

The summary display area of the Summarize Form presents different information depending on your Summary-Expression and whether your summary type is Event or State.

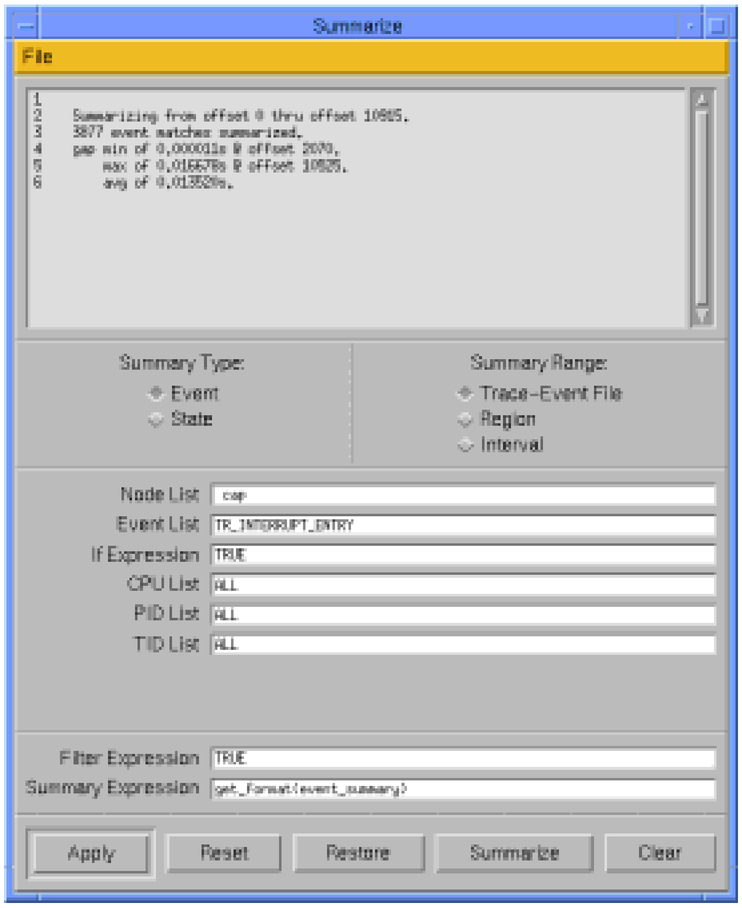
Event Summaries

Configuring *event summaries* is similar to configuring DataBox display objects. The configuration parameters for an event summary are identical to those that are used to configure a DataBox display object. See “DataBox” on page 8-13 for information on how to configure a DataBox.

By default, the Summary-Expression for an event summary type, displays one line for each of the following in the summary display area:

- The range of ordinal trace event numbers (offsets) summarized
- 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

Figure 10-4 shows an event summary.



```

Summarizing from offset 0 thru offset 10925.
3877 event matches summarized.
gap min of 0.000011s @ offset 2070,
    max of 0.016678s @ offset 10925,
    avg of 0.013520s.
```

Summary Type: [✓ Event] State

Summary Range: [✓ Trace-Event File] Region Interval

Node List	cap
Event List	TR_INTERRUPT_ENTRY
If Expression	TRUE
CPU List	ALL
PID List	ALL
TID List	ALL
Filter Expression	TRUE
Summary Expression	get_format(event_summary)

Figure 10-4. The Event Summarize Form

NOTE

The `Node List` field appears in this dialog only when NightTrace is configured to use an RCIM to timestamp events.

State Summaries

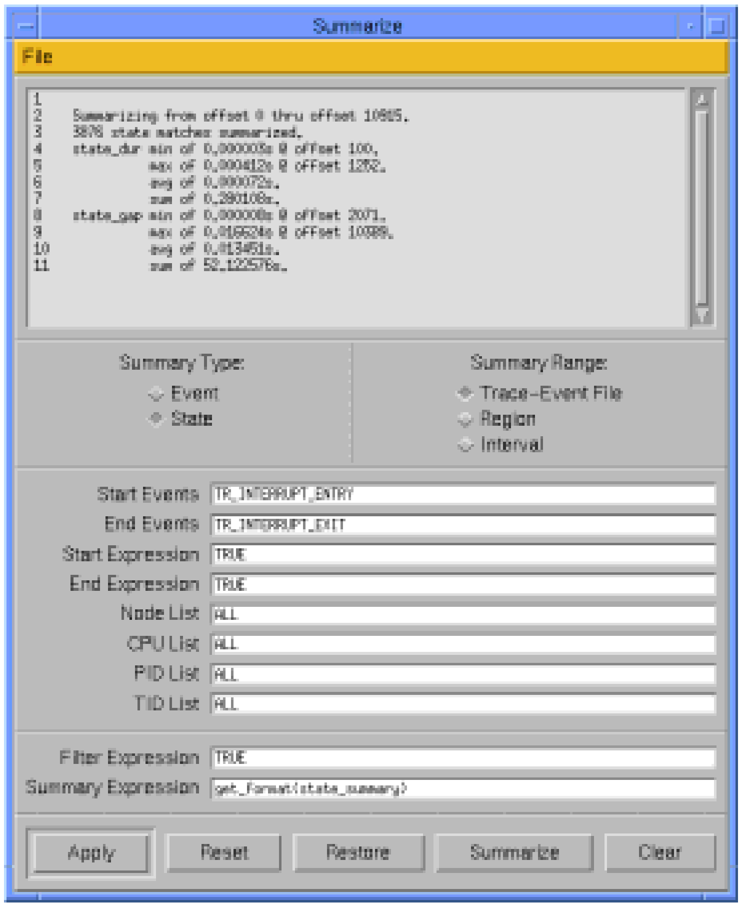
Configuring *state summaries* is similar to configuring StateGraph display objects. The configuration parameters for a state summary are identical to those that are used to configure a StateGraph display object. See “StateGraph” on page 8-14 for information on how to configure a StateGraph.

The main difference between a state summary and a StateGraph is that a state summary shows information textually and a StateGraph shows it graphically.

By default, the `Summary-Expression` for a state summary type, displays one line for each of the following in the summary display area:

- The range of ordinal trace event numbers (offsets) summarized
- 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

Figure 10-5 shows a state summary.



```

Summarizing from offset 0 thru offset 10885.
3878 state matches summarized.
state_dur min of 0.000003s @ offset 100.
          max of 0.000412s @ offset 1252.
          avg of 0.000072s.
          sum of 0.280108s.
state_gap min of 0.000008s @ offset 2071.
          max of 0.018624s @ offset 10388.
          avg of 0.013451s.
          sum of 52.122576s.
```

Summary Type: Event [✓ State]

Summary Range: [✓ Trace-Event File] Region Interval

Start Events	TR_INTERRUPT_ENTRY
End Events	TR_INTERRUPT_EXIT
Start Expression	TRUE
End Expression	TRUE
Node List	ALL
CPU List	ALL
PID List	ALL
TID List	ALL
Filter Expression	TRUE
Summary Expression	get_format(state_summary)

Figure 10-5. The State Summarize Form

NOTE

The Node List field appears in this dialog only when NightTrace is configured to use an RCIM to timestamp events.

Exercise: Using the Search Tool

The following exercise has you search for trace events you logged in "Exercise: Logging Trace Events" on page 4-27, while using files you created in "Exercise: Displaying Trace Events" on page 5-36.

Invoke the NightTrace display utility with the **log** trace event file, the **map** event-map file, and the **page** configuration file.

```
$ ntrace log map page
```

After the display page appears, press the Refresh push button at the bottom right of the page. The current time line should now be positioned inside the first visible state. What is the tag of the current trace event?

```
NAP START
```

Now, bring up the Search tool by clicking on Tools ➪ Search ...

Use the default settings to search globally forward for all trace events and make the interval scroll the current time to the trace event. After one search, what is the tag of the current trace event?

```
NAP_END
```

Keep searching forward until you reach the end of the trace. You should continue to see alternating `NAP_START` and `NAP_END` trace event tags.

Close the Search Form by clicking on the Close push button.

Exercise: Using the Summarize Tool

The following exercise has you summarize trace events you logged in “Exercise: Logging Trace Events” on page 4-27.

While still in **`ntrace`**, bring up the Summarize tool by clicking on Tools ➪ Summarize ...

Press the Summarize button for the default event summary.

How many matches were summarized?

```
22
```

At which offset does the largest gap occur?

```
2
```

How large is this gap?

```
about 1.8 seconds
```

Close the Summarize Form by clicking on File ➪ Close.

This exercise continues in “Exercise: Kernel Tracing” on page 11-38.

11
Tracing the Kernel

11
Tracing the Kernel

Overview

This chapter provides a description of the trace points logged by the kernel. It also discusses the steps required to produce a highly detailed picture of kernel activity with the KernelTrace and NightTrace tools. This lets you customize the default **`ntrace`** kernel display pages or combine kernel information with user-application trace information.

Recommended Reading

The following manuals and documents explain many of the concepts briefly discussed in this chapter.

Table 11-1. Recommended Reading

Manual Title	Concepts
PowerMAX OS Programming Guide	Understanding the kernel and exceptions related to signals
PowerMAX OS Real-Time Guide	Using **`ktrace(1)`** and understanding the kernel and hardclock interrupts
System Administration Volume 1 and Volume 2	Understanding interrupts and creating system devices
Section 2 system manual pages	Understanding system calls
PowerPC 604 RISC Microprocessor User's Manual	Understanding exceptions
HN6200 Architecture Reference Manual	Understanding interrupts
HN6800 Architecture Reference Manual	Understanding interrupts

Using KernelTrace with NightTrace

Using the KernelTrace package is optional. The following steps are involved:

- (Required) Installing the `trace` package on your system. See “Installing Software” on page 2-2.
- (Required) Enabling kernel tracing and default kernel trace points. See “Configuring the Kernel” on page 2-3 and “Default Kernel Trace Points” on page 11-2.
- (Required) Logging kernel trace events. This may also include enabling and disabling more kernel trace points and analyzing textual trace event summaries. See “Kernel Tracing with ktrace” on page 11-8.
- (Optional) Converting a KernelTrace trace event file to a NightTrace trace event file for subsequent graphical analysis with **`ntrace`**. See “Converting KernelTrace Trace Event Files with ntfilter” on page 11-21.
- (Optional) Graphically analyzing trace event summaries. See “Viewing Converted KernelTrace Trace Event Files with ntrace” on page 11-22.

Kernel trace points identify interrupts, exceptions, system calls, context switches, and I/O to various devices. When kernel tracing is enabled, the kernel tests whether to log one or more kernel trace events for each enabled kernel trace point. If the **`ktrace`** tool is running, kernel trace event logging takes place. Understanding the kernel trace points is important for analyzing **`ktrace`** output and creating and modifying graphical kernel display pages for **`ntrace`**.

Default Kernel Trace Points

The file <**`sys/ktrace.h`**> identifies all kernel and device driver trace points. Of these kernel trace points, **`ktrace`** enables only the following by default:

- `TR_SWITCHIN`
- `TR_INTERRUPT_ENTRY` and `TR_INTERRUPT_EXIT`
- `TR_EXCEPTION_ENTRY` and `TR_EXCEPTION_EXIT`
- `TR_SYSCALL_ENTRY`
- `TR_IO_VNODE`
- `TR_ALT_INT_DISPATCH`
- `TR_PROCESS_NAME`

These default kernel trace points are required to get meaningful kernel performance data in a KernelTrace trace event file. However, these trace points are not the only trace points that you will see with **`ntrace`** after converting a KernelTrace trace event file into Night-Trace trace event file format with **`ntfilter`**. Specifically, the following trace points are introduced:

- `TR_SYSCALL_EXIT`
- `TR_SYSCALL_SUSPEND` and `TR_SYSCALL_RESUME`
- `TR_EXCEPTION_SUSPEND` and `TR_EXCEPTION_RESUME`

When **`ntfilter`** converts a KernelTrace trace event file into a NightTrace trace event file, it removes the `TR_IO_VNODE`, `TR_ALT_INT_DISPATCH`, and `TR_PROCESS_NAME` trace events. `TR_ALT_INT_DISPATCH` events are converted to appropriate `TR_INTERRUPT_ENTRY` and/or `TR_INTERRUPT_EXIT` trace points. For each `TR_PROCESS_NAME` event, **`ntfilter`** extracts the process name from the event and adds the name to its process name table which is subsequently written to the **`vectors`** file.

The following sections discuss the trace events that you will see in **`ntrace`** as a result of logging the default kernel trace points.

Context Switch Trace Event

There is only one context switch trace event:

`TR_SWITCHIN` *arg1*

This trace event is logged whenever a process has been switched in and is ready to be run on a specific CPU. Because only one process can run on a given CPU at a time, this trace event also signifies that the process that was running on the CPU immediately prior to the context switch trace event has been switched out and can no longer run. This trace event has one argument:

arg1 The numeric 32-bit global process identifier (PID) of the process being switched in. This information is redundant, since it is identical to the PID that is already associated with the trace event. A PID of `0` indicates that the CPU is idle.

The 32-bit global process identifier uniquely identifies the running process on the system. This identifier is identical to the return value of the `_lwp_global_self()` system call. See “pid()” on page 9-22.

Interrupt Trace Events

There are two trace events associated with interrupts:

`TR_INTERRUPT_ENTRY` *arg1 arg2 arg3*

This trace event is logged whenever an interrupt is entered. It has three arguments:

arg1 The interrupt vector number that indicates the type of interrupt. This is an index into the `vector` string table that is contained within the **`vectors`** file generated by the **`ntfilter`** tool. For more information about the `vector` string table, see “Kernel String Tables” on page 11-32. For more information about the **`ntfilter`** tool, see “Converting KernelTrace Trace Event Files with ntfilter” on page 11-21.

arg2 The interrupt nesting level used by the pre-defined kernel pages to graph the different heights associated with the nesting level. This argument will be `1` for the first interrupt, `2` for a second interrupt that interrupted the first interrupt, `3` for a third interrupt that interrupted the second interrupt, etc.

arg3 The interrupt vector number of the previous interrupt that this interrupt entry is interrupting, if any.

`TR_INTERRUPT_EXIT` *arg1 arg2 arg3*

This trace event is logged whenever an interrupt is exited. Its arguments are identical to those of the `TR_INTERRUPT_ENTRY` trace event.

Exception Trace Events

There are four trace events associated with exceptions:

`TR_EXCEPTION_ENTRY` *arg1*

This trace event is logged whenever an exception is entered. It has one argument:

arg1 The exception vector number that indicates the type of exception. This is an index into the `vector` string table that is contained within the **`vectors`** file generated by the **`ntfilter`** tool. For more information about the `vector` string table, see “Kernel String Tables” on page 11-32. For more information about the **`ntfilter`** tool, see “Converting KernelTrace Trace Event Files with ntfilter” on page 11-21.

`TR_EXCEPTION_SUSPEND` *arg1*

This trace event is logged whenever an exception is suspended by a context switch. It has one argument that is identical to the argument logged with the `TR_EXCEPTION_ENTRY` trace event.

`TR_EXCEPTION_RESUME` *arg1*

This trace event is logged whenever an exception is resumed (i.e., the process that caused the exception to occur, which was switched out before the exception could be completed, is switched back in). A `TR_EXCEPTION_RESUME` trace event will always follow a `TR_EXCEPTION_SUSPEND` event, unless the process is being switched in for the first time since kernel tracing began.

It is possible for several `TR_EXCEPTION_SUSPEND`—`TR_EXCEPTION_RESUME` trace event pairs to occur if the process is switched in and out several times before the exception completes.

The `TR_EXCEPTION_RESUME` trace event has one argument that is identical to the argument logged with the `TR_EXCEPTION_ENTRY` trace event.

`TR_EXCEPTION_EXIT` *arg1*

This trace event is logged whenever an exception is completed. It has one argument that is identical to the argument that is logged with the `TR_EXCEPTION_ENTRY` trace event.

Syscall Trace Events

There are four trace events associated with syscalls:

`TR_SYSCALL_ENTRY` *arg1 arg2 arg3*

This trace event is logged whenever a syscall is entered. It has three arguments:

arg1	This argument is always zero for historical reasons.
arg2	The syscall number that identifies the syscall. This is an index into the pre-defined `syscall` string table.
arg3	The device number that indicates the type of device that is associated with the syscall, if any. This is an index into the pre-defined `device` string table.

For more information about the pre-defined `syscall` and `device` string tables, see “Kernel String Tables” on page 11-32.

`TR_SYSCALL_SUSPEND` *arg1 arg2 arg3*

This trace event is logged whenever a syscall is suspended by a context switch. It has three arguments that are identical to the arguments logged with the `TR_SYSCALL_ENTRY` trace event.

`TR_SYSCALL_RESUME` *arg1 arg2 arg3*

This trace event is logged whenever a syscall is resumed (i.e., the process that caused the syscall to occur, which was switched out before the syscall could be completed, is switched back in). A `TR_SYSCALL_RESUME` trace event will always follow a `TR_SYSCALL_SUSPEND` trace event, unless the process is being switched in for the first time since kernel tracing began.

It is possible for several `TR_SYSCALL_SUSPEND—TR_SYSCALL_RESUME` trace event pairs to occur if the process is switched in and out several times before the syscall completes.

The `TR_SYSCALL_RESUME` trace event has three arguments that are identical to the arguments logged with the `TR_SYSCALL_ENTRY` trace event. However, if a `TR_SYSCALL_RESUME` trace event does not follow a `TR_SYSCALL_SUSPEND` trace event (i.e., it is the first syscall trace event logged by the process since kernel tracing began) *arg2* identifies the syscall as “`can't determine`.”

`TR_SYSCALL_EXIT` *arg1 arg2 arg3*

This trace event is logged whenever a syscall is completed. It has three arguments that are identical to the arguments logged with the `TR_SYSCALL_ENTRY` trace event.

Shared Interrupt Trace Event

The `TR_ALT_INT_DISPATCH` trace point assists in determining the real identity of shared interrupts. A shared interrupt handler invokes another interrupt handler which either directly processes the interrupt or determines more information about the interrupt and dispatches yet another handler to process the interrupt. Because shared interrupts share an interrupt vector, the vector number is not sufficient to distinguish the devices that share a vector. Whenever a shared interrupt handler invokes a handler which actually processes the interrupt, the kernel logs a `TR_ALT_INT_DISPATCH` event to provide the information necessary to uniquely identify the device which generated the interrupt.

When **`ntfilter`** converts a KernelTrace trace event file into a NightTrace trace event file, `TR_ALT_INT_DISPATCH` events are converted to appropriate `TR_INTERRUPT_ENTRY` and/or `TR_INTERRUPT_EXIT` trace points.

Process Name Trace Event

The `TR_PROCESS_NAME` trace point increases the likelihood of successfully associating a process name with a process ID during analysis of trace events in NightTrace. When **`ktrace`** is used to collect kernel trace data, it scans **`/proc`** once during initialization and once during termination to gather process names. However, if the workload to be measured is run after **`ktrace`** has been initialized and completes before **`ktrace`** terminates, that workload will not have a process name available since the process(es) did not exist during either of the **`/proc`** traversals. The kernel will log a `TR_PROCESS_NAME` event, which contains a global LWP ID and either the name of the LWP (if it is available) or the name of the LWP's containing process, in the following instances:

- The kernel creates an LWP, i.e. the kernel creates a kernel daemon.
- A user process creates an LWP either directly through the **`_lwp_create(2)`** system call or indirectly through the Threads Library.
- A user process creates a child process through any of the **`fork(2)`** system calls.
- A user process overlays the existing process with a new process image through any of the **`exec(2)`** system calls.

When **`ntfilter`** converts a KernelTrace trace event file into a NightTrace trace event file, **`ntfilter`** extracts the process name from each `TR_PROCESS_NAME` event and adds the name to its process name table which is subsequently written to the **`vectors`** file.

Kernel Trace Points Not Enabled By Default

There are several kernel trace points which are not enabled by default but two of them deserve special mention. These two events allow you to determine areas in your application code where address faults are occurring, to minimize such faults, and thus improve the application's performance. The following sections discuss the page fault and protection fault kernel trace points.

Page Fault Event

There is one page fault trace event:

`TR_PAGEFLT_ADDR` *arg1 arg2 arg3*

This trace event is logged whenever a kernel or user page fault occurs. The page fault can be either on a data address or on an instruction address. This trace event is not enabled by default because, depending upon system activity, page faults may occur reasonably frequently. This trace event has three arguments:

arg1	The data address which caused the page fault. If the page fault occurred on an instruction, this will be set to zero.
arg2	The program counter value at the time of the page fault.
arg3	The flag indicating whether the fault occurred on a kernel address or on a user address. A value of zero indicates that the fault occurred on a user address. A value of one indicates that the fault occurred on a kernel address.

Protection Fault Event

There is one protection fault trace event:

`TR_PROTFLT_ADDR` *arg1 arg2 arg3*

This trace event is logged whenever a kernel or user protection fault occurs. The protection fault can be either on a data address or on an instruction address. This trace event is not enabled by default because, depending upon system activity, protection faults may occur reasonably frequently. This trace event has three arguments:

arg1	The data address which caused the protection fault. If the protection fault occurred on an instruction, then this will be set to zero.
arg2	The program counter value at the time of the protection fault.
arg3	The flag indicating whether the fault occurred on a kernel address or on a user address. A value of zero indicates that the fault occurred on

a user address. A value of one indicates that the fault occurred on a kernel address.

Kernel Tracing with ktrace

The KernelTrace feature consists of the **ktrace(1)** and **ntfilter(1)** tools. Use **ktrace** to collect KernelTrace data and generate textual summaries. Then use **ntfilter** to convert KernelTrace trace event files from **ktrace** into NightTrace trace event files, suitable for **ntrace** graphical displays. (See “Converting KernelTrace Trace Event Files with ntfilter” on page 11-21 for details.)

ktrace collects data about the execution time of interrupts, exceptions, system calls, context switches, and I/O to various devices. The **ktrace** program uses **/dev/trace** to enable trace points within the kernel. These trace points cause trace records to be logged to kernel trace buffers. Filling the kernel trace buffers causes trace records to be written to an output file. (For more information about these buffers, see <**sys/ktrace.h**>.) The trace record includes a trace event identifier, a timestamp that corresponds to the time at which the kernel event occurred, and some additional system information.

ktrace can be used to read kernel trace buffers or an input file and to analyze the log of trace records. When **ktrace** is used to analyze trace records, it prints a summary that contains the average, minimum, and maximum times for interrupts, exceptions, system calls, context switches, and I/O calls to various devices.

CAUTION

Summaries may be inaccurate because:

Kernels built with kernel tracing enabled run slower than those built without it.

One CPU may block another CPU from writing to a trace buffer, causing the time to record a trace point on a multiprocessor system to be imprecise.

The kernel allocates buffers of three pages each (12,288 bytes) to **ktrace**. This is part of the kernel’s initialized global data, meaning these are reserved physical pages.

Normally **ktrace** does not lose kernel trace events. If **ktrace** issues an error message about lost trace events, ask your system administrator to increase the size of `TR_BUFFER_COUNT` in **/etc/conf/mtune.d/trace** by running the **idtune(1M)** command, rebuilding, and rebooting the system. (Usually a `TR_BUFFER_COUNT` of 5 is sufficient.) For more information about tunable parameters, see “Tunable Parameters” in *System Administration Volume 2*.

The trace mechanism is not able to deal with losing events. This is because it needs to match up start events with end events (interrupt start and interrupt end, for example) in order to produce meaningful statistics. There are other ordering dependencies too. Therefore, if you see something like the following message:

```
ERROR: events lost 15773
fatal error
```

ktrace exits immediately afterwards. Note that if you are logging data to a file, the data written is still valid (the buffers with corrupted data will not have been written to the file) so any summary produced should still be meaningful.

Invoking ktrace

The **ktrace** kernel trace logging and analysis tool resides on your system under **/usr/bin/ktrace**. You can override some default functionality by invoking **ktrace** with options. The full **ktrace** invocation syntax is:

ktrace [**-help**] [**-version**] [**-measure**] [**-output** *file*]
[**-bufferwrap** *count*] [**-disable**][**-enable** *tracepoint*]
[**-priority** *priority*][**-clock** *source*][**-cpu** *CPU*]
[**-process** *PID*][**-input** *file*] [**-ticks**] [**-wall**]
[**-start** *sec*] [**-nohardclock**][**-raw**] [**-verbose**]

The following sections discuss the **ktrace** options and arguments.

ktrace Options

You can abbreviate all **ktrace** options to their shortest unambiguous length, but most of the examples in this manual use the long option name. These options are case-insensitive.

ktrace options include:

-help	Display the **ktrace** invocation syntax on standard output and exit. Screen 11-1 shows an example.

```
usage:  ktrace  [-help]  [-version]  [-measure]  [-output file]
                [-bufferwrap count]  [-disable]  [-enable tracepoint]
                [-priority priority]  [-clock source]  [-cpu CPU]
                [-process PID]  [-input file]  [-ticks]  [-wall]
                [-start sec]  [-nohardclock]  [-raw]  [-verbose]

General options:
   -help                  Write this message to standard output
   -version               Write current ktrace version to standard output
   -measure               Measure the time required to log a trace event

Options for collection of kernel trace data:
   -output file           File to write collected data to
   -bufferwrap count      Write last count buffers to the output file
   -disable               Disable all default kernel trace points
   -enable tracepoint     Enable specified kernel trace point
   -priority priority     Run ktrace at specified RT priority (default: max)
   -clock source          Specify source of event time stamps
      Valid values for source are:
         default             Use the default system clock
         rcim_tick           Use the RCIM synchronized tick clock

Options for analysis of kernel trace data:
   -cpu CPU               Include in analysis trace events only for given CPU
                          (default: all CPUs)
   -process PID           Include in analysis trace events only for given PID
                          (default: all PIDs)
   -input file            File of data to analyze (default: /dev/trace)
   -ticks                 Report time in ticks instead of elapsed time
   -wall                  Use wall times for the summary calculations
   -start sec             Exclude from analysis trace events before given time
   -nohardclock           Exclude from analysis hardclock interrupts
   -raw                   Display raw data for each trace event
   -verbose               Display verbose data for each trace event
```

Screen 11-1. Sample Output from the ktrace -help Option

`-version` Display the current **`ktrace`** version stamp on standard output and exit.

`-measure` Display to standard output the time required to log a trace event and exit.

The following options are for the collection of kernel trace data.

`-output` *file* Write raw trace data to KernelTrace trace event file *file* rather than writing a summary to standard output.

`-bufferwrap` *count* Write only the last *count* trace buffers of the most recent trace events to the output file. By default, all trace events are written to the output file. Each trace buffer contains `TR_BUFFER_SIZE` trace events (for the value of this define, please see **`/usr/include/sys/ktrace.h`**).

CAUTION

Using the **`-bufferwrap`** option may cause a process name to be unavailable for a process ID during subsequent trace data analysis. If the `TR_PROCESS_NAME` event which names the process is overwritten by buffer wraparound and the process name is not picked up by **`ktrace`** during its two scans of **`/proc`**, the process will not have a name available for it.

`-disable` Disable all default kernel trace points. (See "Default Kernel Trace Points" on page 11-2 for details.) Use a subsequent **`-enable`** to enable individual trace points.

`-enable` *tracepoint* Enable kernel trace point *tracepoint* in addition to the default kernel trace points. (For information about the default kernel trace points, see "Default Kernel Trace Points" on page 11-2.). The allowed values for *tracepoint* appear in the include file <**`sys/ktrace.h`**>.
You can disable and enable all trace points at trace time. You would usually do this if you are interested only in tracking context switches and do not want to incur the overhead of the other trace points. For example, invoke **`ktrace`** as follows:

```
ktrace -disable -enable 50
```

where `50` is the ID of the `TR_SWITCHIN` trace point.

`-priority` *priority* Run **`ktrace`** at the specified real-time priority. The default is the maximum real-time priority.

`-clock` *source* Use the specified *source* for trace event timestamps. The *source* is required. Valid *source* values are:

`default`

the interval timer (NightHawk 6000 Series) or the Time Base Register (Power Hawk/PowerStack)

`rcim_tick`

the RCIM synchronized tick clock

NOTE: If you specify **`rcim_tick`** for the *source* and the system on which you are tracing does not have an RCIM installed or configured or if the RCIM synchronized tick clock on the system on which you are tracing is stopped, **`ktrace`** will exit with an error.

The following options are for the analysis of kernel trace data.

`-cpu` *CPU* Include in analysis trace events only for logical CPU *CPU*. The default is all CPUs.

`-process` *PID*	Include in analysis trace events only for process *PID*, where *PID* is the global process identifier of the process (e.g., `237'1`). See "PID List" on page 8-7 and "Context Switch Trace Event" on page 11-3 for more information about global process identifiers. The default is all PIDs. Out of necessity, the trace-point data includes interrupt trace events for interrupts that occurred while *PID* was running and excludes interrupts generated by *PID*'s I/O requests. From the NightTrace perspective, kernel trace points can be thought of as being logged by the processes that are running on the system's CPUs, <u>not</u> by the kernel. For example, even though a `TR_SYSCALL_ENTRY` trace event is logged by the kernel, the PID that is associated with the trace event is the PID of the process that made the syscall. The PID of an exception entry is the PID of the process that caused the exception to occur, and the PID of an interrupt is the PID of the process that was interrupted by the interrupt. The PID of a context switch is the PID of the process that is being switched in. Thus, a context switch can be thought of as the first trace event that a process logs when it is ready to start running.
`-input` *file*	Read or analyze data in *file*. The default is to open the device, **`/dev/trace`**, that collects the trace-point data from the running kernel.
`-ticks`	Report time in ticks (256-nanosecond increments of the interval timer) instead of elapsed time. This can be useful for correlating times with other tools.
`-wall`	Make all time calculations as wall-time calculations. Unlike default summaries, include time blocked in the kernel and time spent in interrupts that preempt execution of the current process or current interrupt. By default, the time reported for system calls, I/O calls, and other exceptions only includes time spent actually processing the system call, I/O call, or other exception. Also by default, the time reported for an interrupt would include only time spent while actually processing the current interrupt.
`-start` *sec*	Exclude from analysis trace events logged during the first *sec* seconds of elapsed time.
`-nohardclock`	Exclude from analysis trace events for hardclock interrupts.
`-raw`	Display the raw trace records with minimal formatting. Note that this option creates large output files. See "ktrace -raw Listing" on page 11-20 for an example.

-verbose	Display the raw trace records with data interpretation and maximal formatting, including logical CPU ID, PID, and elapsed time for all trace points. Note that this option creates huge output files. See “ktrace -verbose Listing” on page 11-19 for an example.

To collect raw trace data from a running kernel in a KernelTrace trace event file named **raw_klog**, run **ktrace** with the invocation:

```
$ ktrace -output raw_klog
```

While **ktrace** is running, run the workload to be measured. Terminate **ktrace** by sending it a <Ctrl> <c> or using the **kill(1)** command to send it a SIGINTR signal.

CAUTION

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

The KernelTrace trace event files created with **ktrace** have unique magic numbers, and can be identified with the **file(1)** command. For example:

```
$ file raw_klog
raw_klog:   KernelTrace trace event file
```

To run **ktrace** using the KernelTrace trace event file **raw_klog** as input:

```
$ ktrace -input raw_klog > summary_file
```

The preceding example analyzes the trace records saved in **raw_klog** and writes textual time summaries of kernel activity to **summary_file**.

KernelTrace trace event files are not compatible with the NightTrace tool set. However, these files may be converted to NightTrace trace-file format with the **ntfilter** tool. See “Converting KernelTrace Trace Event Files with ntfilter” on page 11-21 for more information.

For more information about kernel tracing see the PowerMAX OS *Real-Time Guide*.

Viewing KernelTrace Trace Event Files with ktrace

Once you have a **ktrace** KernelTrace trace event file, you can use **ktrace** to analyze and display it. You can use options to modify the summary reports.

ktrace Kernel Activity Summaries

By default **ktrace** produces textual summaries of kernel activity. Screen 11-2 through Screen 11-6 illustrate a summary written to standard output after running **ktrace** for a while without any options and interrupting it with a <Ctrl> <c>.

Configuration Summary

Screen 11-2 shows an example **ktrace** invocation and the resulting configuration summary.

```
Configuration Summary
=====================

Run date (data gathering):         Thu Aug 26 17:12:41 1999
Total run time:                    10.267777 sec
Machine type/node:                 Power Hawk 640/henry
Kernel type/release:               PowerMAX_OS/4.3
Event time stamp source:           RCIM synchronized tick clock
Ktrace options (data gathering):   ktrace -output /tmp/k1 -clock rcim_tick
Ktrace options (analysis):         ktrace -input /tmp/k1 -verbose
```

Screen 11-2. Configuration Summary

A configuration summary shows:

- The date of the trace
- The total elapsed wall time in seconds of the trace
- The machine type and name
- The operating system type and release level
- The event timestamp source
- The options used to invoke **ktrace**

System Call Summary

Screen 11-3 shows an example of a **ktrace** system call summary.

```
System Call Summary
===================

system call      count      min        @time        avg        max        @time
--------------- ------ -------- ------------ ---------- ---------- ------------
fork                13     2534     7.445815       5089      10139     5.134528
read              3661       60    19.864994        180       1299    29.369101
write             3535       53    13.312549         98       6711    13.777209
open                70      171     7.523197        370        978     4.822754
close              176       21     5.180711         51        404     7.719841
creat                2      707     7.539468        708        710     7.449845
chdir                2      224     7.627435        228        232     7.375272
gtime                8       18     7.290165         24         46    10.464009
chmod                1      416     5.006350        416        416     5.006350
chown                1      492     5.005885        492        492     5.005885
brk                 49       29     7.660741         95        186     5.272032
lseek               41       29     5.284300         37         54     5.157868
getpid              14       22     7.290425         24         27     7.669769
setuid               1      101     7.326856        101        101     7.326856
getuid               8       18     7.396601         24         28     7.396569
alarm               10       23     7.306740         56         88     5.176025
pause                1      285     7.513753        285        285     7.513753
access              18      221     7.405962        542       1615    10.456365
nice                 1       61     5.291145         61         61     5.291145
setpgrp              3       20     7.389734         53        111     5.144212
dup                  3       29     5.183234         37         46     5.183189
pipe                 2      447     7.663649        460        472     7.443250
setgid               1       64     7.326563         64         64     7.326563
getgid               4       17     7.396665         19         21    13.231724
ssig                98       26     7.306903         50        105    13.232873
...
```

Screen 11-3. System Call Summary

A system call summary contains one line for each type of system call that occurred during the trace. Each line includes the following information:

- The number of times the system call occurred
- The minimum duration in microseconds of the system call and the elapsed time in seconds during the trace when it occurred
- The average duration in microseconds of all occurrences of the system call
- The maximum duration in microseconds of the system call and the elapsed time in seconds during the trace when it occurred

Exception and Interrupt Summaries

Screen 11-4 shows examples of **`ktrace`** exception and interrupt summaries.

```
Exception Summary
=================

exception        count      min        @time        avg        max        @time
--------------- ------ -------- ------------ ---------- ---------- ------------
inst access        484       23    13.229954         38        675    20.004900
data access       3645       22     4.880435        176       1001     5.034892
syscall           8556       17     7.396665        161      13363     4.871685


Interrupt Summary
=================


cpu/interrupt    count      min        @time        avg        max        @time
--------------- ------ -------- ------------ ---------- ---------- ------------
1/spurious int       6       26    17.933975        155        197    29.369312
0/delayed int       41       36    13.416419        106        151    13.285586
1/rescheduling      24       13     5.252413         23         49    13.538494
0/rescheduling       8       11    10.463875         19         57     7.452930
1/eg               118       75    12.811796        187        359     5.465960
0/hsa             1069       12     9.220024         49        152    16.816370
0/softclock         17       34    16.816488         69        112    13.766340
0/console          493       28    14.145049         64        267    29.367308
1/hardclock       1762        9     2.898720         15         81     5.132768
0/hardclock       1761       23    16.349611         82        957     6.666530
1/cross proc       183       21    15.479378         27         52     4.998217
0/cross proc      1040       20    18.382416         24         65    13.648542
```

Screen 11-4. Exception and Interrupt Summaries

An exception summary contains one line for each type of exception that occurred during the trace. An interrupt summary contains one line for each type of interrupt that occurred on a specific CPU during the trace. Each line includes the following information:

- The number of times the exception or interrupt occurred
- The minimum duration in microseconds of the exception or interrupt and the elapsed time in seconds during the trace when it occurred
- The average duration in microseconds of all occurrences of the exception or interrupt
- The maximum duration in microseconds of the exception or interrupt and the elapsed time in seconds during the trace when it occurred

Exception and Interrupt Total Time Summaries

Screen 11-5 shows examples of **`ktrace`** exception and interrupt total time summaries.

```
Exception Total Times
=====================

exception              total time spent in exception
---------------      ----------------------------------
inst access              0.018562 sec  (     18562 usec)
data access              0.643099 sec  (    643099 usec)
syscall                  1.380817 sec  (   1380816 usec)

per cpu                total time spent in exceptions
---------------      ----------------------------------
cpu 0                    0.334344 sec  (    334344 usec)
cpu 1                    1.708132 sec  (   1708132 usec)


Interrupt Total Times
=====================

cpu/interrupt          total time spent in interrupt
---------------      ----------------------------------
0/delayed int            0.004361 sec  (      4361 usec)
0/rescheduling           0.000155 sec  (       155 usec)
0/hsa                    0.053209 sec  (     53209 usec)
0/softclock              0.001184 sec  (      1184 usec)
0/console                0.031591 sec  (     31591 usec)
0/hardclock              0.145527 sec  (    145527 usec)
0/cross proc             0.026075 sec  (     26075 usec)

1/spurious int           0.000931 sec  (       931 usec)
1/rescheduling           0.000560 sec  (       560 usec)
1/eg                     0.022186 sec  (     22186 usec)
1/hardclock              0.027925 sec  (     27925 usec)
1/cross proc             0.005143 sec  (      5143 usec)

per cpu                total time spent in interrupts
---------------      ----------------------------------
cpu 0                    0.258850 sec  (    258850 usec)
cpu 1                    0.056358 sec  (     56358 usec)
```

Screen 11-5. Exception and Interrupt Total Time Summaries

The exception total time summary includes:

- A line showing the total time spent in each type of exception
- The total time spent in all exceptions per CPU

The interrupt total time summary includes:

- A line showing the total time spent in each type of interrupt per CPU
- The total time spent in all interrupts per CPU

Device Summary

Screen 11-6 shows a **`ktrace`** device summary.

```
Device Summary
==============

open            count      min        @time        avg        max       @time
------------- ------ -------- ------------ ---------- ---------- ------------
file            189      262    14.019941       2661       7879     7.600755
dir              24      125    13.425103       1528       3089    13.213366

close          count      min        @time        avg        max        @time
------------- ------ -------- ------------ ---------- ---------- ------------
file            216       28     7.410805         76       1518     5.029229
dir              38       30     5.593385         51         82    13.360715
fifo             44       37     6.572858        113        379     6.010460
pts               7       79     9.673261        105        123     7.432435
nullzero          2       56    10.338360         57         57    10.338130

read           count      min        @time        avg        max        @time
------------- ------ -------- ------------ ---------- ---------- ------------
file           1181       58    10.350096        201        629    10.607500
fifo             50       75    10.426975        436        776    14.009368
mm                2      111     4.574140        124        138    14.574760
trace            21      150     6.008959        207        241    13.851000
tcp               3      107     4.586716        130        153    14.588139

write          count      min        @time        avg        max        @time
------------- ------ -------- ------------ ---------- ---------- ------------
file            138      190     9.666454        528       4005    10.293841
fifo             46      108    10.417129        447        808     1.691544
ptm               1      207    17.909775        207        207    17.909775
pts               6      322     6.007895        369        456    14.008412
tcp               8      566    16.009590        638        704     6.009530

ioctl          count      min        @time        avg        max        @time
------------- ------ -------- ------------ ---------- ---------- ------------
file             33       35     5.712818         47         70     7.457112
fifo              9       64    15.274572        109        236    14.011427
udp              96       67     3.108406         78        123     3.140494
pts              26       39     5.305794         72        122     5.707121
tcp             114       39     4.566289         55        123    14.587958

poll           count      min        @time        avg        max        @time
------------- ------ -------- ------------ ---------- ---------- ------------
fifo              5      319     9.991041        371        455    15.991276
ptm               8       81    17.909548        291        380    10.008830
udp               8      200     5.480585        226        268     3.386693
tcp              61       68     6.008808         93        340    14.587762
ticotsor         23      401     6.115654        505        680     9.132074
```

Screen 11-6. Device Summary

A device summary includes a line of statistics for each type of device accessed by an `open`, `close`, `read`, `write`, `ioctl` or `poll` system call during the trace. The line contains the same information that is included in the system call summary for those system calls except that here it is further broken down by device type.

ktrace Trace Event Listings

The preceding summaries are good for getting an overall idea of system activity, but in general you also want to be able to examine the kernel trace events that occurred around the maximum times for certain durations. In order to do this you need to invoke **`ktrace`** in a two step process:

```
$ ktrace -output raw_klog
locking into memory
setting priority to RT 59
open /dev/trace
initialize
gather trace point data
<Ctrl> <c>
terminating
$ ktrace -input raw_klog > summary_file
```

Thus, an exhaustive log of kernel activity is recorded in the **`raw_klog`** output file.

TIP:

The output KernelTrace trace event file must be on a local file system and not an NFS file system. Check the destination file system first if **`ktrace`** always seems to be losing numerous trace events.

ktrace -verbose Listing

By default, **`ktrace`** produces statistical summaries like the ones shown on the preceding screens. You can use the **`-verbose`** option to produce a verbose listing of all events occurring in the run. For example,

```
$ ktrace -input raw_klog -verbose > listing
```

produces something like Screen 11-7.

```
 16637   12.373326  cpu=0 pid=sbc_msgd    171  hardclock entry
    32   12.373358  cpu=0 pid=sbc_msgd    171  hardclock exit        32u
 16665   12.373361  cpu=1 pid=idle        163  softclock entry
     3   12.373365  cpu=1 pid=idle        163  softclock exit        3u
 16634   12.389993  cpu=0 pid=sbc_msgd    171  hardclock entry
    29   12.390023  cpu=0 pid=sbc_msgd    171  hardclock exit        29u
 16660   12.390025  cpu=1 pid=idle        163  softclock entry
     4   12.390029  cpu=1 pid=idle        163  softclock exit        4u
 16636   12.406660  cpu=0 pid=sbc_msgd    171  hardclock entry
    41   12.406701  cpu=0 pid=sbc_msgd    171  hardclock exit        41u
 16674   12.406703  cpu=1 pid=idle        163  softclock entry
     3   12.406707  cpu=1 pid=idle        163  softclock exit        3u
  8408   12.415110  cpu=0 pid=sbc_msgd    178  decintr entry
  8431   12.415139  cpu=1 pid=idle        160  rescheduling entry
    46   12.415156  cpu=0 pid=sbc_msgd    178  decintr exit          46u
   183   12.415333  cpu=1 pid=idle        160  rescheduling exit     194u
    12   12.415346  cpu=1 pid=in.rlogind  context switch
    20   12.415366  cpu=1 pid=in.rlogind          mip
    12   12.415379  cpu=1 pid=in.rlogind          ptm
    39   12.415419  cpu=1 pid=in.rlogind   12  syscall exit          116u poll
    11   12.415431  cpu=1 pid=in.rlogind  context switch
    23   12.415455  cpu=1 pid=in.rlogind   12  syscall entry
     2   12.415457  cpu=1 pid=in.rlogind          read
```

Screen 11-7. ktrace -verbose Listing

The number in the second column of each line in the listing is the elapsed time of the trace event in seconds since tracing began. Notice that trace events from different CPUs and different processes are intermixed in this listing.

The numbers to the left of the trace event descriptions are the actual vector numbers of the trace events and are generally not of interest. A number in parentheses to the right of a trace event description indicates a nesting level; the nesting level is displayed only if the level is greater than one. After an exit of a syscall, interrupt, or exception, the duration in microseconds of the elapsed time from the matching start is given. Finally, for a syscall exit, the name of the system call exiting is listed.

ktrace -raw Listing

Use the **`-raw`** option if you suspect that **`ktrace`** is producing inaccurate listings or summaries. For example,

```
$ ktrace -input raw_klog -raw > listing
```

produces something like Screen 11-8.

```
1172: code=0050, cpu=1, spins=0000, time=93162472, param=00000004
1173: code=0013, cpu=0, spins=0000, time=931624c7, param=00000049
1174: code=0012, cpu=1, spins=0000, time=93162508, param=00000048
1175: code=0079, cpu=1, spins=0000, time=00000000, param=00004800
1176: code=0013, cpu=1, spins=0000, time=9316255d, param=00000048
1177: code=0050, cpu=1, spins=0000, time=931626ae, param=ffffffff
1178: code=0012, cpu=0, spins=0000, time=931720e3, param=00000049
1179: code=0079, cpu=0, spins=0000, time=00000000, param=00004900
1180: code=0013, cpu=0, spins=0000, time=93172194, param=00000049
1181: code=0012, cpu=1, spins=0000, time=931721d1, param=00000048
1182: code=0079, cpu=1, spins=0000, time=00000000, param=00004800
1183: code=0013, cpu=1, spins=0000, time=93172205, param=00000048
1184: code=0012, cpu=0, spins=0000, time=93181f36, param=00000049
1185: code=0079, cpu=0, spins=0000, time=00000000, param=00004900
1186: code=0013, cpu=0, spins=0000, time=93181fc2, param=00000049
1187: code=0012, cpu=1, spins=0000, time=93181ffc, param=00000048
1188: code=0079, cpu=1, spins=0000, time=00000000, param=00004800
1189: code=0013, cpu=1, spins=0000, time=93182030, param=00000048
1190: code=0012, cpu=0, spins=0000, time=93191d8a, param=00000049
1191: code=0079, cpu=0, spins=0000, time=00000000, param=00004900
1192: code=0013, cpu=0, spins=0000, time=93191e0c, param=00000049
1193: code=0012, cpu=1, spins=0000, time=93191e45, param=00000048
1194: code=0079, cpu=1, spins=0000, time=00000000, param=00004800
1195: code=0013, cpu=1, spins=0000, time=93191e71, param=00000048
1196: code=0012, cpu=0, spins=0000, time=931a1be0, param=00000049
1197: code=0079, cpu=0, spins=0000, time=00000000, param=00004900
1198: code=0013, cpu=0, spins=0000, time=931a1c69, param=00000049
```

Screen 11-8. ktrace -raw Listing

In a raw listing, no information is interpreted by **`ktrace`**. The leading number is the offset of the trace event in the kernel trace event buffer where the trace event was logged and is generally not of interest.

The fields for each trace event are labeled clearly. The `spins` field indicates how much contention existed between multiple CPUs logging trace events at the trace event logging time. The `code`, `cpu`, and `spins` fields are displayed in decimal, and the `time` and `param` fields are displayed in hexadecimal.

Converting KernelTrace Trace Event Files with ntfilter

The KernelTrace feature consists of the **`ktrace(1)`** and **`ntfilter(1)`** tools. Use **`ktrace`** to collect kernel trace event data and generate textual summaries. (See “Kernel Tracing with ktrace” on page 11-8 for details.) Then use **`ntfilter`** to convert KernelTrace trace event files from **`ktrace`** into NightTrace trace event files, suitable for **`ntrace`** graphical displays.

The usual way to convert a KernelTrace trace event file into NightTrace trace event file format is to invoke **`ntfilter`** in the following manner:

`ntfilter -v` < *raw_klog* > *klog*

where:

`-v`	Causes **`ntfilter`** to produce a file named **`vectors`** in the current directory as it converts a KernelTrace trace event file into NightTrace trace event file format. The **`vectors`** file contains the definition of the `vector`, `syscall`, and `pid` string tables. See “Pre-Defined String Tables” on page 5-15 and “Kernel String Tables” on page 11-32 for more information about these string tables.
raw_klog	The KernelTrace trace event file to be converted.
klog	The converted KernelTrace trace event file in NightTrace trace event file format.

For large KernelTrace trace event files, the conversion process may take several seconds to complete. The resulting NightTrace trace event file is approximately twice the size of the KernelTrace trace event file.

Converted KernelTrace trace event files are accepted as input to the **`ntrace`** display utility. A **`vectors`** file created with the **`-v`** option should always be specified to **`ntrace`** along with its corresponding converted KernelTrace trace event file. The **`vectors`** file is generated dynamically because it is system-configuration dependent. Without a **`vectors`** file, **`ntrace`** will not be able to display the names of the system processes, interrupts, and exceptions that occurred during kernel tracing.

Even though the vector information is coded into the KernelTrace trace event files created by **`ktrace`**, the system you run **`ntfilter`** on must be the same as the system that created the KernelTrace trace event file due to size differences in statically-allocated arrays internal to **`ntfilter`**.

The **`file(1)`** command can be used to verify that the KernelTrace trace event file has been converted. For example:

```
$ file klog
klog:       NightTrace trace event file
```

There is no difference between a converted KernelTrace trace event file and a NightTrace trace event file, other than the fact that the converted KernelTrace trace event file contains kernel trace events.

Viewing Converted KernelTrace Trace Event Files with ntrace

All of the kernel trace event tags are defined in the **/usr/lib/NightTrace/eventmap** file. This file is automatically read by **ntrace** at start-up time.

Once you have a converted KernelTrace trace event file, you can use **ntrace** to examine it. You may design your own display pages to view converted KernelTrace trace event files; see Chapter 7 and Chapter 8 for more information. Alternatively, you may use and/or modify pre-defined kernel display pages. These pages are discussed in the following sections.

Kernel Display Pages

Figure 11-1 shows the File menu of the **ntrace** Global Window. This menu contains a Default Kernel Page menu item which is used to open a dynamically-built kernel display page. The Default Kernel Page menu item is enabled only if a converted KernelTrace trace event file has been supplied to **ntrace** on the command line.

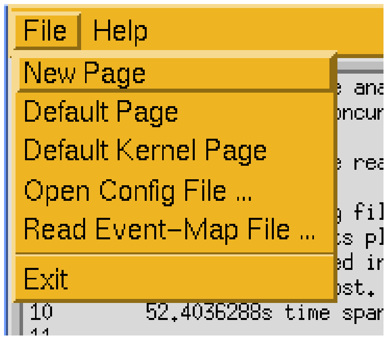
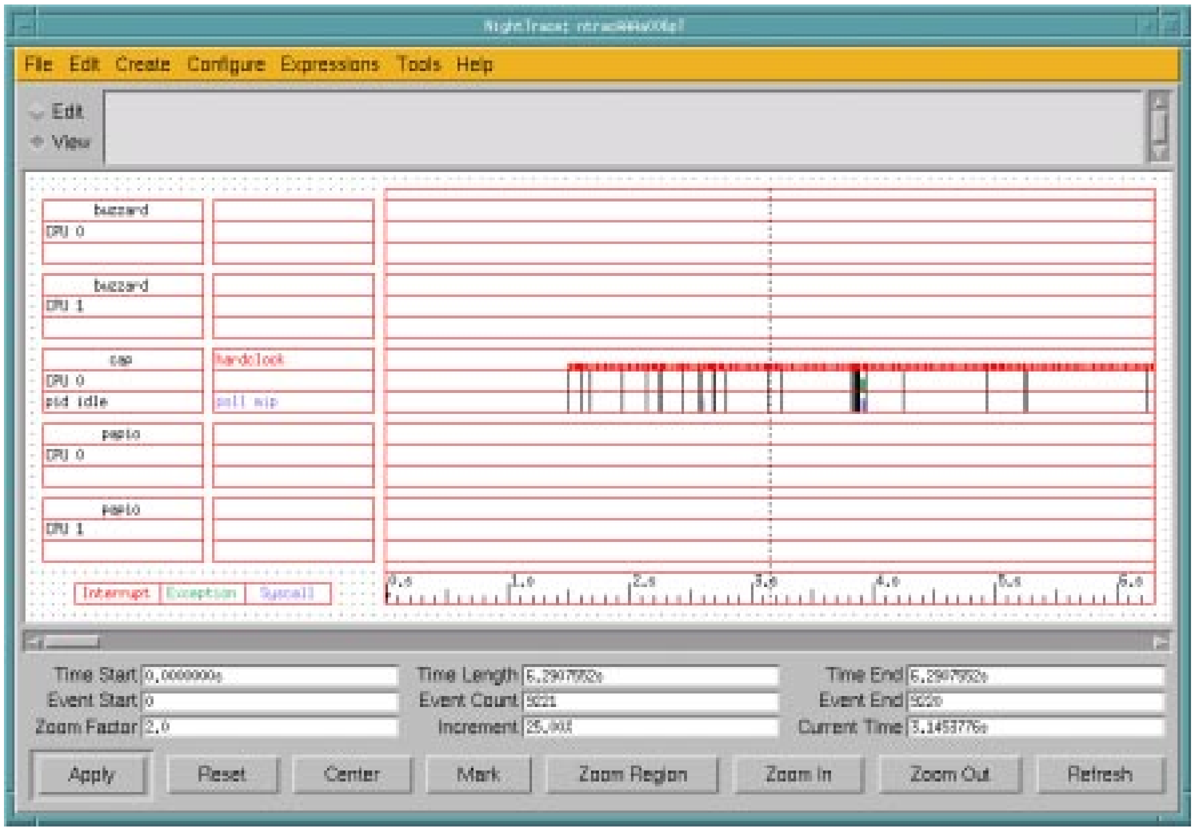


Figure 11-1. Global Window File Menu

Figure 11-2 shows a sample kernel display page in View mode constructed from trace files on two different *nodes*.

NOTE

The node information is displayed only when NightTrace is configured to use an RCIM to timestamp events.



Time Start 0.0000000s
Event Start 0
Zoom Factor 2.0
Time Length 6.2907952s
Event Count 5221
Increment 25.00%
Time End 6.2907952s
Event End 5220
Current Time 3.1453776s

Figure 11-2. Sample Kernel Display Page

`ntrace` determines the number of CPUs on the system from information in the converted KernelTrace trace event file.

RCIM Default Kernel Display Page

When viewing KernelTrace trace event files that have been timestamped by the RCIM tick clock, **`ntrace`** determines the number of distinct nodes/hosts which have trace files and constructs a default display page accordingly. When you create a default kernel display page from trace event files that have been timestamped by the RCIM tick clock, **`ntrace`** pops up a dialog box that allows you to select the nodes you wish to display on that kernel page.

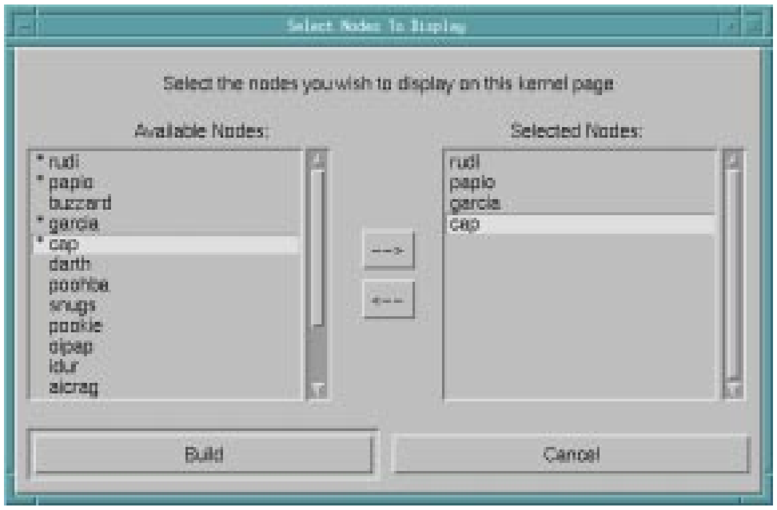


Figure 11-3. Node Selection Dialog

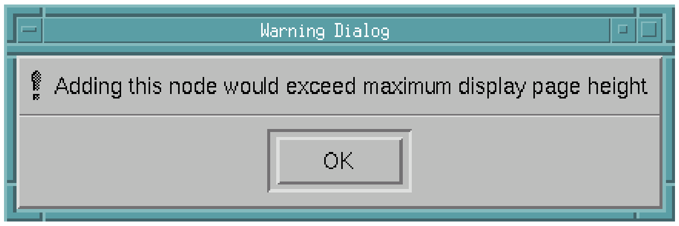
The Available Nodes list shows all nodes that NightTrace has found in the trace files. The Selected Nodes list contains all nodes you want shown on the kernel display page you are building.

NOTE

> An asterisk (*) next to a node in the Available Nodes list indicates that the particular node has already been selected through the Node Selection Dialog.

You may select the nodes you wish to be included on the kernel display page you are building by either double-clicking each node name in the Available Nodes list or by selecting a node from that list and using the right arrow button to add it to the list of Selected Nodes. When the list of Selected Nodes contains all the nodes you wish to display on your kernel display page, you may press the Build button.

As each node is added to the list of Selected Nodes, **ntrace** figures out how much vertical real estate the grid needs (based upon the number of nodes you wish to display and how many CPUs each node has). If the required vertical space does not exceed the maximum grid height, **ntrace** will allow the page to be created. Otherwise, **ntrace** will pop up a warning dialog window and will not allow the page to be created.



Warning Dialog

Adding this node would exceed maximum display page height

OK

Figure 11-4. Node Selection Warning Dialog

Figure 11-5 shows the display of information for a CPU on a particular node on a dynamically-built kernel page.

NOTE

The node information is displayed only when NightTrace is configured to use an RCIM to timestamp events.

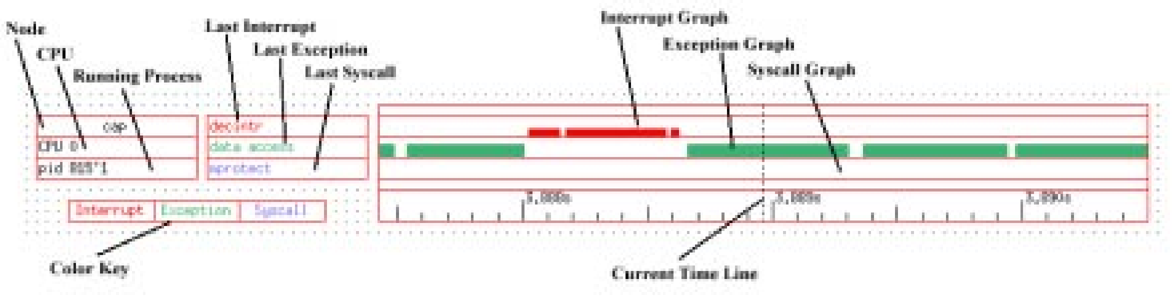


Figure 11-5. Per-CPU Information

There are several pieces of information being displayed for each CPU. The position of the current time line determines the values that appear on the kernel display pages. Moving the current time line within the current interval does not change the graphical displays. However, the textual displays always reflect the last values prior to the current time line.

The following sections discuss all of the different pieces of information in detail.

CPU Information

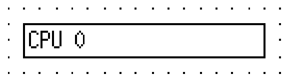


Figure 11-6. CPU Box

Figure 11-6 shows a CPU box. The CPU box simply identifies which logical CPU the displayed data corresponds to. Logical CPU numbers are related to, but not necessarily identical to, physical CPU numbers.

Each CPU in a system has a four-bit physical CPU number. The physical CPU number is dependent on which card slot the CPU card containing the CPU is in and which location on the card the CPU is in. The low two bits of the number specify the location on the card that the CPU is in. These bits are either 00 for the first CPU location or 01 for the second. The high two bits of the physical CPU number contain the CPU card slot number. These bits can be 00, 01, 10, or 11 (or, in decimal, 0, 1, 2, or 3).

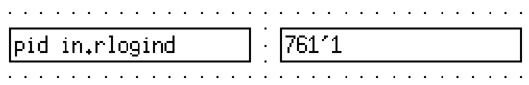
For simplicity, most kernel utilities translate the physical CPU numbers into logical CPU numbers. The mapping is accomplished by listing the physical CPU numbers of all configured CPUs in ascending order and then numbering them sequentially, starting with zero. For example, a four-CPU system having two CPUs on a card in slot 1 and two CPUs on a card in slot 3 will have physical CPU numbers 4 (0100), 5 (0101), 12 (1100) and 13 (1101). Table 11-2 shows the logical CPU mapping of this example system.

Table 11-2. Example Logical CPU Mapping

Physical CPU Number	Logical CPU Number
4 (0100)	0
5 (0101)	1
12 (1100)	2
13 (1101)	3

The CPU box is a GridLabel display object. See Chapter 7 and Chapter 8 for more information on creating and configuring GridLabels.

Running Process Information



pid in.rlogind

761´1

Figure 11-7. Running Process Boxes

Figure 11-7 shows two examples of running process boxes. The running process box shows the process that is executing at the current time on the associated CPU. The process is listed by name, or by its raw PID and LWPID if no name is available. See “PID List” on page 8-7 for more information about PIDs, raw PIDs and LWPIDs.

You can supply NightTrace trace event files to **`ntrace`** along with converted Kernel-Trace trace event files. **`ntrace`** uses the process names of all processes that logged trace events when displaying the running process.

The running process box is a DataBox display object. See Chapter 7 and Chapter 8 for more information on creating and configuring DataBoxes.

Node Information

Figure 11-8. Node Box

Figure 11-7 shows a node box. The node box simply identifies which node the displayed data corresponds to.

NOTE

The node information is displayed only when NightTrace is configured to use an RCIM to timestamp events.

The node box is a GridLabel display object. See Chapter 7 and Chapter 8 for more information on creating and configuring GridLabels.

Context Switch Information

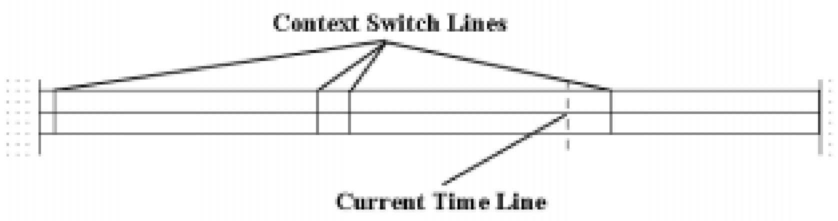


Figure 11-9. Context Switch Lines

Figure 11-9 shows an example of several context switch lines. *Context switch lines* are superimposed on the exception and syscall graphs. They indicate that the kernel has switched out the process that was previously running on the CPU and switched in a new process. There is a direct correlation between context switch lines and the running process box: the running process box shows the process associated with the context switch line that immediately precedes the current time line.

Interrupt Information

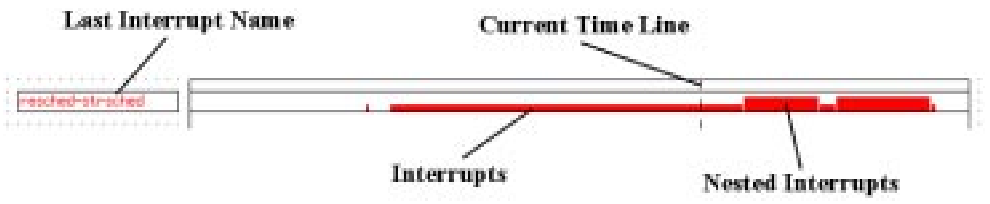


Figure 11-10. Last Interrupt Box and Interrupt Graph

Figure 11-10 shows a last interrupt box and an interrupt graph. The interrupt graph displays a state that is drawn whenever an interrupt is executing on the associated CPU. Interrupts can be interrupted while executing, and the interrupt graph shows this interrupt nesting by increasing the height of the state bar. Although interrupts can nest, all interrupts must complete before the process they interrupt can be switched out. Therefore, you will never see a context switch occur in the middle of an interrupt.

The last interrupt box displays the name of the last interrupt prior to the current time line that executed (and may still be executing) on the associated CPU. It can be used with the interrupt graph to identify any interrupts that are currently visible on the graph. Simply move the current time line onto a graphed interrupt, and the last interrupt box will update to display the name of the interrupt.

Because the last interrupt box displays the name of the last interrupt that executed, it is possible for there to be no interrupts visible on the interrupt graph even though the last interrupt box contains a valid interrupt name. This just signifies that the last interrupt on the CPU ended prior to the beginning of the current interval.

An interrupt that is seen very often is the hardclock interrupt, which usually accounts for 15% of the total number of trace events logged by the kernel. If you are not interested in hardclock interrupts, they can be ignored by **`ntrace`**, improving performance and readability. See “ntrace Options” on page 5-3 for more information.

The last interrupt box is a DataBox display object, and the last interrupt graph is a DataGraph display object. See Chapter 7 and Chapter 8 for more information on creating and configuring DataBoxes and DataGraphs.

Exception Information

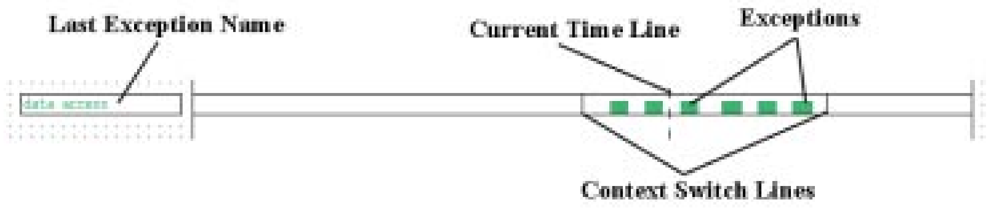


Figure 11-11. Last Exception Box and Exception Graph

Figure 11-11 shows a last exception box and an exception graph. The exception graph displays a state that is drawn whenever an exception is executing on the associated CPU. Unlike interrupts, exceptions cannot nest, so they are always graphed with the same height.

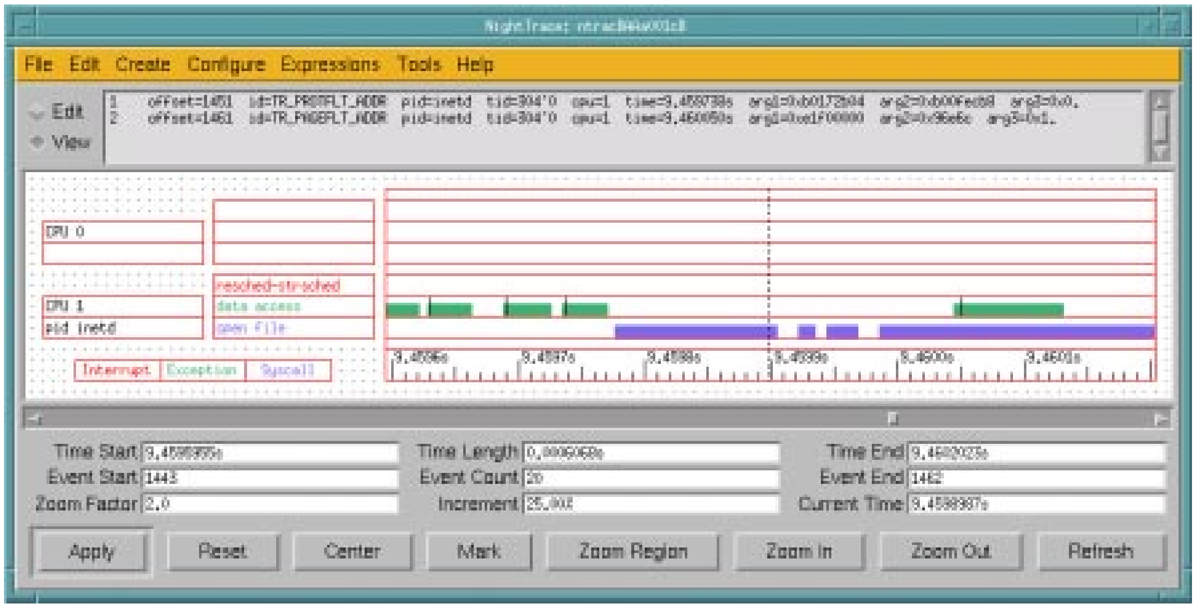
Context switch lines are superimposed on exception graphs. It is common to see a context switch line at what looks like the very end (or beginning) of an exception. Usually, this does not indicate that the exception has ended, only that it has been suspended because the process that originated the exception has switched out. The exception resumes when the process is switched back in again. An example of an exception being suspended and resumed can be seen at the left end of the exception graph in Figure 11-11.

The last exception box displays the last exception prior to the current time line that executed (and may still be executing) on the associated CPU. It can be used with the exception graph to identify any exceptions that are currently visible on the graph. Simply move the current time line onto a graphed exception, and the last exception box will update to display the name of the exception.

Because the last exception box displays the name of the last exception that executed, it is possible for there to be no exceptions visible on the exception graph even though the last exception box contains a valid exception name. This just signifies that the last exception on the CPU ended prior to the beginning of the current interval.

The last exception box is a DataBox display object, and the last exception graph is a State-Graph display object. See Chapter 7 and Chapter 8 for more information on creating and configuring DataBoxes and StateGraphs.

Lines indicating `TR_PAGEFLT_ADDR` and `TR_PROTFLT_ADDR` events are also superimposed on exception graphs. Exception graphs display these trace points to allow you to obtain a formatted dump of them in the message display area by clicking on the events with mouse button 2. An example of a `TR_PAGEFLT_ADDR` and a `TR_PROTFLT_ADDR` event as well as their associated data in the message display area can be seen in Figure 11-12.



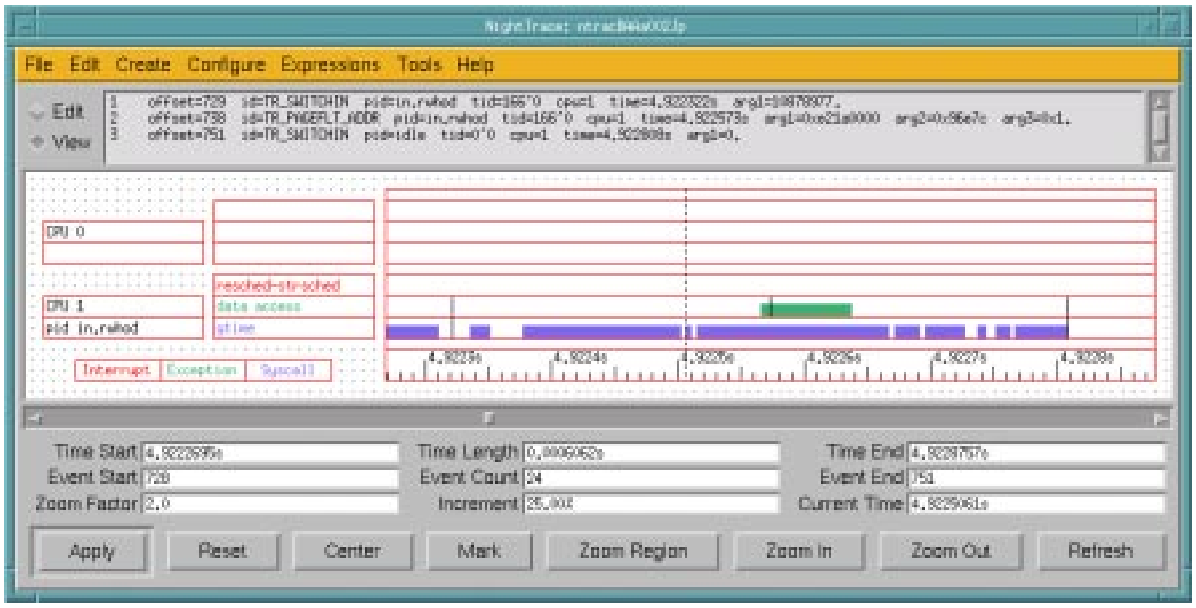
```
1    offset=1451  id=TR_PROTFLT_ADDR  pid=inetd  tid=304'0  cpu=1  time=9.459738s  arg1=0xb0172b04  arg2=0xb00fed68  arg3=0x0.
2    offset=1461  id=TR_PAGEFLT_ADDR  pid=inetd  tid=304'0  cpu=1  time=9.460050s  arg1=0xe1f00000  arg2=0x96e6c  arg3=0x1.
```

Time Start	9.4595955s	Time Length	0.0006068s	Time End	9.4602023s
Event Start	1443	Event Count	20	Event End	1462
Zoom Factor	2.0	Increment	25.00%	Current Time	9.4598987s

Figure 11-12. TR_PAGEFLT_ADDR and TR_PROTFLT_ADDR Events

Note the `TR_PROTFLT_ADDR` event to the left of the current time line at `time`=9.459738 and the `TR_PAGEFLT_ADDR` event to the right of the current time line at `time`=9.460050 and the corresponding data in the message display area. (See "The Display Page" on page 7-2 for more information on the message display area and other elements of the display page.)

Note also that the `TR_PROTFLT_ADDR` and `TR_PAGEFLT_ADDR` events are represented by a vertical line that only intersects the exception state graph whereas a `TR_SWITCHIN` event (see "Context Switch Trace Event" on page 11-3) intersects both the exception and syscall state graphs. In addition, `TR_PROTFLT_ADDR` and `TR_PAGEFLT_ADDR` events will only appear within a currently executing exception. This can be seen in Figure 11-13.



```
1    offset=729  id=TR_SWITCHIN  pid=in.rwhod  tid=166'0  cpu=1  time=4.922322s  arg1=0x878977.
2    offset=738  id=TR_PAGEFLT_ADDR  pid=in.rwhod  tid=166'0  cpu=1  time=4.922573s  arg1=0xe21a8000  arg2=0x96e7c  arg3=0x1.
3    offset=751  id=TR_SWITCHIN  pid=idle  tid=0'0  cpu=1  time=4.922808s  arg1=0.
```

Time Start 4.9222695s Time Length 0.0006062s Time End 4.9228757s
Event Start 728 Event Count 24 Event End 751
Zoom Factor 2.0 Increment 25.00% Current Time 4.9225061s

Figure 11-13. TR_SWITCHIN vs. TR_PAGEFLT_ADDR and TR_PROTFLT_ADDR Events

Syscall Information

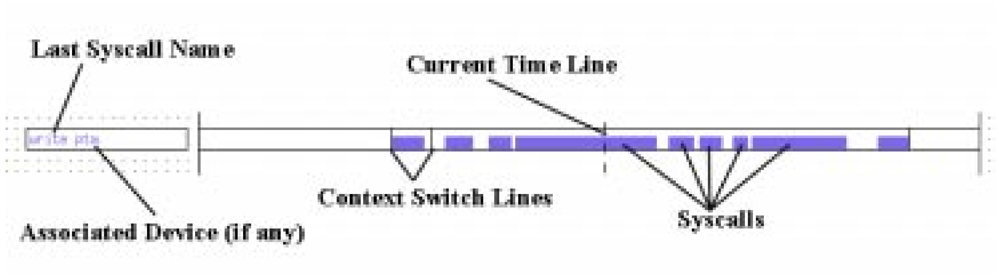


Figure 11-14. Last Syscall Box and Syscall Graph

Figure 11-14 shows a last syscall box and a syscall graph. The syscall graph displays a state that is drawn whenever a system call (syscall) is executing on the associated CPU. Unlike interrupts, syscalls cannot nest, so they are always graphed with the same height.

Context switch lines are superimposed on syscall graphs. It is common to see a context switch line at what looks like the very end (or beginning) of a syscall. Usually, this does not indicate that the syscall has ended, only that it has been suspended because the process that originated the syscall has switched out. The syscall resumes when the process is switched back in again. An example of a syscall being suspended and resumed can be seen at the right end of the syscall graph in Figure 11-14.

The last syscall box displays the last syscall prior to the current time line that executed (and may still be executing) on the associated CPU. If the syscall is associated with a device, the name of the device is shown after the name of the syscall.

The last syscall box can be used with the syscall graph to identify any syscalls that are currently visible on the graph. Simply move the current time line onto a graphed syscall, and the last syscall box will update to display the name of the syscall.

Because the last syscall box displays the name of the last syscall that executed, it is possible for there to be no syscalls visible on the syscall graph even though the last syscall box contains a valid syscall name. This just signifies that the last syscall on the CPU ended prior to the beginning of the current interval.

It is possible for the first syscall logged by a process since kernel tracing began to be unknown. This can occur if the process is switched in and immediately resumes a syscall that was previously suspended. If this occurs, the last syscall box will display "`can't determine`" for the name of the syscall.

The last syscall box is a DataBox display object, and the last syscall graph is a StateGraph display object. See Chapter 7 and Chapter 8 for more information on creating and configuring DataBoxes and StateGraphs.

Color Information

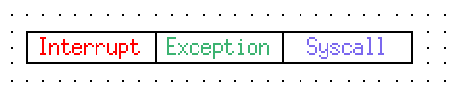


Figure 11-15. Color Key

Figure 11-15 shows the color key that is located on the bottom left of the grid on the pre-defined kernel display pages. The color key is useful only on X terminals that support more colors than just black and white.

The text in the color key is color-coded. By default, the word "`Interrupt`" is red, and all display objects on the kernel display page that display information about interrupts are also red. By default, the word "`Exception`" is green, and all display objects that display information about exceptions are also green. By default, the word "`Syscall`" is blue, and all display objects that display information about syscalls are also blue.

The default colors of the different groups of kernel objects can be controlled with X resources. The colors are specified on a per-CPU basis. The default resources for logical CPU 0 are:

```
Ntrace*Color*GridObject*interrupt0*foreground: red
Ntrace*Color*GridObject*exception0*foreground: green
Ntrace*Color*GridObject*syscall0*foreground:   blue
```

See Appendix B for more information on X resources.

Kernel String Tables

There are seven kernel related pre-defined string tables. They are:

`vector`

This string table contains the interrupt and exception vector names associated with the system that the kernel tracing was performed on. It is contained in the **`vectors`** file created by the **`ntfilter`** tool. For more information on creating a **`vectors`** file, see “Converting KernelTrace Trace Event Files with ntfilter” on page 11-21. For brief descriptions of the entries in the `vector` string table, see “Interrupts” on page 11-35 and “Exceptions” on page 11-36.

This table is indexed by an exception/interrupt vector number or an exception/interrupt vector name. Examples of using this table are:

```
get_string(vector, arg3())
get_string(vector, 15)
get_item(vector, “ncr_intr”)
```

`syscall`

This string table contains the names of all the possible syscalls that can occur on the system. It is contained in the **`vectors`** file created by the **`ntfilter`** tool. For brief descriptions of the entries in the `syscall` table, see “Syscalls” on page 11-37.

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. **`ktrace`** gathers this information from the **`/etc/conf/node.d`** directory on the current system and places it into the KernelTrace trace event file. It is transferred to the **`vectors`** file created by the **`ntfilter`** tool.

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 created by the **`ntfilter`** tool. 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 created by the **`ntfilter`** tool.

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 created by the **`ntfilter`** tool.

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 created by the **`ntfilter`** tool.

This table is indexed by a device number or a device name. Examples of using this table are:

```
get_string(device_simulator1, arg3())
get_string(device_simulator4, 3604484)
get_item(device_controller, “rtc”)
```

The `pid` string table is also used by the kernel display pages. For more information on the `pid` string table, see “Pre-Defined String Tables” on page 5-15. For examples of function calls with these tables, see Table 8-3.

Kernel Reference

The following sections provide a brief reference to the most common interrupts, exceptions, and syscalls.

Interrupts

There are many different types of interrupts that can be logged by the kernel. The possible types are listed in the system-dependent `vector` string table that is generated by the **`ntfilter`** tool. There are two main categories of interrupts:

- Non-device-related interrupts
- Device-related interrupts

The members of these two categories are described in the following two sections.

Non-Device-Related Interrupts

Table 11-3 provides an alphabetical list of the most common non-device-related interrupts.

Table 11-3. Non-Device-Related Interrupt Reference

Interrupt	Description
callout int	A real time clock interrupt that is used internally by the kernel.
console wake	An interrupt caused by the console wakeup button.
int on no int	An interrupt that occurs during the processing of another interrupt.
power fail	A power fail interrupt.
rescheduling	A rescheduling interrupt used to trigger a context switch to run the highest priority process that is ready to run.
softclock	An interrupt used to process system callout queue entries.
spurious int	An interrupt that usually indicates an unreported or already-removed interrupt. This interrupt appears only in kernel traces.
sysfault int	An interrupt indicating that a fatal hardware condition has been detected.
user int	A user-level interrupt. See **`iconnect(3C)`** for a description of enabling user-level interrupts.
xcall int	An inter-processor interrupt used for cache flushing, delivering exceptions to another processor, performance monitoring, and halting processors.

For more information about interrupts see **`intstat(1M)`** and **`uistat(1M)`**.

Device-Related Interrupts

The names printed for device interrupts correspond to the device names in the system configuration files. See *System Administration Volume 2* for information on adding devices to a system.

Table 11-4 provides an alphabetical list of the most common device-specific interrupts. For more information on a device-specific interrupt, refer to the documentation associated with the particular device.

Table 11-4. Device-Related Interrupt Reference

Interrupt	Description
consintr	A console terminal interrupt.
eg	An Eagle ethernet controller interrupt.
eti_intr	An edge-triggered interrupt.
ex	An Excellan ethernet controller interrupt.
gpib	An IEEE-488 GPIB controller interrupt.
hardclock	A 60-Hertz clock interrupt.
hd	An HDC disk-controller interrupt.
hps	An HPS serial line-controller interrupt.
hrm	A reflective memory interrupt.
hsa	An HSA disk controller interrupt.
hsd	An HSD controller interrupt.
ie	An integral ethernet interrupt.
is	An integral SCSI controller interrupt.
mpcc	An MPCC controller interrupt.
pgintr	An FDDI controller interrupt.
rtcintr	A real-time clock interrupt.
xy	A Xylogics tape-controller interrupt.

Exceptions

There are many different types of exceptions that can be logged by the kernel. The possible types are listed in the system-dependent `vector` string table that is generated by the **ntfilter** tool. Table 11-5 is an alphabetical list of the most common exceptions. See the *PowerPC 604 RISC Microprocessor User's Manual* for more information.

Table 11-5. Exception Reference

Exception	Description
data access	An exception indicating that a page fault for a data page occurred.
decrementer	An exception that occurs when the decrementer register counts down to zero.
float unavail	An exception that occurs the first time a process attempts to use the floating-point unit.
inst access	A page fault exception that occurs during an instruction fetch.
inst brkpt	An exception indicating that a breakpoint instruction was executed.
kstack overflow	A fatal exception generated due to kernel errors.
machine check	A fatal exception generated for various reasons including parity errors, hardware failures, and kernel errors.
misaligned	An exception indicating that a load, store, or exchange instruction was attempted with a destination memory address not consistent with the size of the access.
program	An exception indicating one of several possible conditions including divide by zero, invalid instruction, and privilege violation.
trace	An exception generated during single stepping of the CPU.

Syscalls

There are many different types of syscalls that can be logged by the kernel. The possible types are listed in the architecture-dependent `syscall` string table that is dynamically generated into the **vectors** file. For an up-to-date, alphabetical list and brief description of all syscalls, type in the following command:

```
$ apropos "(2)" | pg
```

For most syscalls grouped by function, see the *Compilation Systems Volume 2 (Concepts)* manual. For more information about a specific syscall, see the associated man page. For information about syscalls in an executable that has not been instrumented with trace points, see **truss(1)**.

Exercise: Kernel Tracing

The following exercise has you log kernel and user trace events with the application you created in “Exercise: Logging Trace Events” on page 4-27.

1. Run the **`ktrace`** program in the background so it creates a KernelTrace trace event file named **`raw_klog`**.
2. Sleep for three seconds to allow for **`ktrace`** to initialize.
3. Invoke the **`ntraceud`** daemon with your **`log`** trace event file.
4. Execute your **`entry_exit`** program.
5. Quit running the **`ntraceud`** daemon.
6. Quit running **`ktrace`**.
7. Sleep for three seconds to allow for **`ktrace`** to shutdown.
8. Use the **`ntfilter`** program to convert the **`raw_klog`** KernelTrace trace event file into a NightTrace file named **`klog`**.

A shell script with the following commands is one possible solution:

```
#!/bin/ksh
ktrace -output raw_klog &
sleep 3
ntraceud log
entry_exit
ntraceud -quit log
kill %1
sleep 3
ntfilter -v < raw_klog > klog
```

A
Performance Tuning

Overview

Although NightTrace's defaults are designed for maximum efficiency, your NightTrace environment and application may have special requirements that warrant some performance tuning. You may want to investigate the following issues:

- Preventing trace event loss
- Ensuring accurate timings
- Optimizing file system and CPU usage
- Conserving disk space
- Conserving memory and accelerating **`ntrace`**

Preventing Trace Events Loss

By default, NightTrace copies all user trace events from the shared memory buffer to the trace event file. This means that normally NightTrace neither discards nor loses trace events.

To conserve disk space, you may invoke **`ntraceud`** with the **`-filewrap`** or **`-buffer-wrap`** option. However, by doing so, you are telling NightTrace to intentionally discard older or less-vital trace events. If discarding trace events is undesirable, run **`ntraceud`** in expansive mode. To do this, invoke **`ntraceud`** without the **`-filewrap`** and **`-buffer-wrap`** options. See "Conserving Disk Space" on page A-4 for more information.

When NightTrace *discards* trace events, it is intentional. When NightTrace *loses* trace events, it is not. NightTrace does not report discarded trace events; it does, however, report lost trace events. Most trace event loss is preventable by flushing the shared memory buffer often.

NightTrace shows trace event loss in the following ways:

- As a non-zero "events lost" statistic from **`ntraceud -stats`** *trace_file*, from **`ntrace -filestats`**, or on the **`ntrace`** Global Window
- As a reverse video "`L`" on the **`ntrace`** display page Ruler at the location where the trace event was lost

If trace event loss seems excessive, you can do the following:

Action	Reason
Decrease **`-cutoff`**, the shared memory buffer-full cutoff percentage for **`ntraceud`**	Increase the chance that the **`ntraceud`** daemon will have enough time to copy the trace events in the shared memory buffer to disk before the shared memory buffer fills up.
Decrease **`-timeout`**, the **`ntraceud`** timeout interval	(Same)
Call `trace_flush()` or `trace_trigger()` often from within your application, especially when your application is at a non-time critical point	(Same)
Increase **`-memsize`**, the shared memory buffer size for **`ntraceud`**	(Same)

Use the following command to see the system settings for the current, default, minimum, and maximum shared memory segment size:

```
$ /etc/conf/bin/idtune -g SHMMAX
```

See the **`idtune(1M)`** man page for more information.

A few other factors can affect trace event loss. Processes in your application may write trace events into the shared memory buffer at the same time that **`ntraceud`** is flushing trace events from the shared memory buffer to the trace event file; if the trace event incoming rate exceeds the flush rate, trace events may not be recorded. Furthermore, when NightTrace must choose between operating unobtrusively and logging all trace events, it favors being unobtrusive.

See Chapter 4 for more information on **`ntraceud`** options and modes. For more information on `trace_flush()` or `trace_trigger()`, see "trace_flush() and trace_trigger()" on page 3-20.

In kernel tracing, **`ktrace(1)`** usually does not lose trace events. If **`ktrace`** issues an error message about lost trace events:

- Verify that the output KernelTrace trace event file is on a local file system and not an NFS file system. If you run the following command and there is a colon (`:`) in the "`Filesystem`" column, the file is on an NFS file system.

  $ **`df`** *kernel_trace_file*

- Ask your system administrator to increase the size of `TR_BUFFER_COUNT` in **`/etc/conf/mtune.d/trace`** by running the **`idtune(1M)`** command, rebuild, and reboot the system. (Usually a `TR_BUFFER_COUNT` of 5 is sufficient.) The kernel allocates buffers of 3 pages each (12,288 bytes) to **`ktrace`**. This is part of the kernel's initialized global data, meaning these are reserved physical pages.

Ensuring Accurate Timings

If you lack the privilege to lock your pages in memory (`P_PLOCK`), you must invoke **`ntraceud`** with the **`-lockdisable`** option. If your application lacks read and write privilege to **`/dev/spl`** you must invoke **`ntraceud`** with the **`-ipldisable`** option. Invoking **`ntraceud`** with either the **`-lockdisable`** or **`-ipldisable`** option, may introduce delays and waiting within your application. Use the **`-lockdisable`** and **`-ipldisable`** options only when necessary. For more information on the **`-lockdisable`** option, see "Option to Prevent Page Locking (-lockdisable)" on page 4-9. For more information on the **`-ipldisable`** option, see "Option to Disable the IPL Register (-ipldisable)" on page 4-8.

By default, **`ntraceud`** and NightTrace library routines use page locking to prevent page faults during trace event logging. NightTrace also modifies the interrupt priority level (IPL) register; this action prevents rescheduling and interrupts during trace event logging. NightTrace prevents the operating system from pre-empting your trace event logging application to make itself most unobtrusive to your application.

If the application must wake the **`ntraceud`** daemon unexpectedly, overhead can cause trace event timings to be distorted. Do one or more of the following to increase the likelihood that the daemon will be awake when needed and to make sure that disk write rates are as fast as the application's logging rate:

- Increase the shared memory buffer size (**`-memsize`**)
- Decrease the shared memory buffer-full cutoff percentage (**`-cutoff`**)
- Decrease the **`ntraceud`** timeout interval (**`-timeout`**)
- Call `trace_flush()` or `trace_trigger()` appropriately

For more information on the **`-memsize`**, **`-cutoff`**, and **`-timeout`** options, and `trace_flush()`, see, respectively, "Option to Define Shared Memory Buffer Size (-memsize)" on page 4-14, "Option to Set the Buffer-Full Cutoff Percentage (-cutoff)" on page 4-16, "Option to Set Timeout Interval (-timeout)" on page 4-15, and "trace_flush() and trace_trigger()" on page 3-20.

Optimizing File System and CPU Usage

Different systems may share files via the Network File System (NFS); however, accessing an NFS-mounted file takes longer than accessing a local file. You get the best NightTrace and KernelTrace performance if you avoid NFS accesses; put your trace event file on the same system where both the **`ntraceud`** daemon (or **`ktrace`**) and your application run. To determine whether your disk is local to your system, verify that it is mounted on **`/dev`** and not on another host. You can do this by running the **`df(1)`** command and looking for a colon (:) in the "`Filesystem`" column.

A single system may have more than one CPU. Consider assigning the **`ntraceud`** daemon (or **`ktrace`**) and your application to different CPUs on the same system; this way, **`ntraceud`** (or **`ktrace`**) does not interfere with your application.

You can use the **mpadvise(3C)** library routine to help you determine which CPUs exist on this system. You can assign **ntraceud** (or **ktrace**) and your application to particular CPUs with the **run(1)** command.

```
$ run -bbias  command
```

Conserving Disk Space

To determine how much disk space is available on your system, run the **df(1)** command with the **-k** option and look at the "avail" column. You can conserve disk space if you permit NightTrace to discard some trace events. To do this, invoke **ntraceud** with either the **-filewrap** option or the **-bufferwrap** option.

The **ntraceud -filewrap** option makes NightTrace operate in file-wraparound mode, rather than in expansive mode. In file-wraparound mode the trace event file can become full of trace events. When this happens, **ntraceud** overwrites the oldest trace events at the beginning of the file with the newest ones. The overwriting is called *discarding trace events*. For more information on file-wraparound mode, see "Option to Establish File-Wraparound Mode (-filewrap)" on page 4-10.

The **ntraceud -bufferwrap** option makes NightTrace operate in buffer-wraparound mode, rather than in expansive mode. When the buffer is full in buffer-wraparound mode, the application treats the shared memory buffer as a circular queue and overwrites the oldest trace events with the newest ones. This overwriting continues until your application explicitly calls `trace_flush()` or `trace_trigger()`. Only then, does **ntraceud** copy the remaining trace events from the shared memory buffer to the trace event file. The overwriting is called *discarding trace events*. For more information on buffer-wraparound mode, see "Option to Establish Buffer-Wraparound Mode (-bufferwrap)" on page 4-11.

By default, **ntraceud** operates in expansive mode, not file-wraparound or buffer-wraparound mode. In expansive mode, NightTrace uses the most disk space because it does not discard any trace events.

You can also conserve disk space by invoking **ntraceud** with the **-disable** option so it logs fewer trace events. For details, see "trace_enable(), trace_disable(), and Their Variants" on page 3-16.

The **-bufferwrap** and **-disable** options to **ktrace** offer similar benefits. The **-cpu** and **-process** options also limit kernel trace event logging. See "ktrace Options" on page 11-9 for details.

Conserving Memory and Accelerating ntrace

ntrace can be a memory-intensive tool. By default, when **ntrace** starts up, it loads <u>all</u> trace event information into memory; therefore, the more trace events in your trace event file(s), the more memory **ntrace** uses. When you move the scroll bar on the Display Page to change the displayed interval, **ntrace** processes all trace events between the last interval and this one; if there are many trace events, the display update (or search) may seem slow. To conserve memory and accelerate **`ntrace`**:

- Log only trace events you are really interested in.
- Invoke **`ntrace`** only with the trace event files that are essential to your analysis.
- Invoke **`ntrace`** with options (**`-nohardclock`**, **`-process -start`**, and **`-end`**) that restrict which trace events get loaded. For more information about **`ntrace`** options, see “ntrace Options” on page 5-3.

B GUI Customization

Overview

The graphical user interface (GUI) for **`ntrace`** is based on OSF/Motif. **`ntrace`** runs in the environment of the X Window System. Your X terminal vendor supplies you with vendor-specific directories and files that pertain to colors and fonts. The file that contains available colors is called **`rgb.txt`**. The fonts that your X server supports are in the **`/usr/lib/X11/fonts`** directory.

`ntrace` has default values for X resources. These resources include fonts, some push button names, window titles, window-component dimensions, and colors. You can override the following default X resource settings by providing new values in the following places:

- In your **`.Xdefaults`** file
- On the **`ntrace`** invocation line
- In a resource file that the **`xrdb(1)`** X resource database manager reads

If you specify the same X resource on the **`ntrace`** invocation line and in your **`.Xdefaults`** file, the setting on the invocation line overrides the one in the file.

An X resource line has the following format:

object`*`*subobject*`[*`*subobject*`...]*`*attribute*`:` *value*

where:

object	Is the name of the X client program, **`Ntrace`**.
subobject	Is a level in the widget (window component) hierarchy with the most general level first; this always begins on an upper-case letter. In **`ntrace`**, the first *subobject* is often `Color` for color displays or `Mono` for monochrome displays. The last *subobject* may be the name of your display object. For more information about display object names, see “Display Object Name” on page 8-4.
attribute	Is a characteristic of the last *subobject*; this always begins on a lower–case letter.
value	Is a setting for the *attribute*.

It is possible to omit levels from the widget hierarchy. If you specify all levels of the widget hierarchy and then a *value*, the value applies to that specific widget. If you leave out levels of the widget hierarchy, the attribute applies more generally, possibly to a class of widgets.

For more information on X resources, see “Recommended Reading” on page 1-7 and the *X Window System User’s Guide*.

Default X-Resource Settings for ntrace

`ntrace`’s default X-resource settings follow. They are primarily grouped by window and display object. There are some subobjects and attributes that appear in many settings. Table B-1 lists several common subobjects and attributes along with their meanings.

Table B-1. Meanings of Common Subobjects and Attributes

Subobject/Attribute	Meaning
`TextScrollbox`	The message (or summary) display area
`Dialog`	The dialog box
`name`	The window title. Any window that has a name attribute also has a geometry attribute.
`geometry`	The location and dimensions of the window. See “Recommended Reading” on page 1-7 for more information.
`open`	A push button name in a File Selection Dialog Box
`caption`	The descriptive text within a window

In the following X-resource strings, default values are shown where they exist.

The resource strings for the global window message display area dimensions and window title are:

```
Ntrace*GlobalWindow*TextScrollbox*defaultLines:  20
Ntrace*GlobalWindow*TextScrollbox*defaultChars:  80
Ntrace*GlobalWindow*name:                         NightTrace
Ntrace*GlobalWindow*geometry:
```

The resource strings for the line count of the display page message area follow. Note: `minimumLines` must be less than or equal to `defaultLines`, and `defaultLines` must be less than or equal to `maximumLines`.

```
Ntrace*DisplayPage*TextScrollbox*defaultLines:  3
Ntrace*DisplayPage*TextScrollbox*maximumLines:  3
Ntrace*DisplayPage*TextScrollbox*minimumLines:  3
```

The resource strings for grid attributes follow. **`ntrace`** uses the `defaultDotsHigh` and `defaultDotsWide` attributes only for new display pages. Note: if `defaultDotsWide` is too narrow to accommodate all the display page push buttons, **`ntrace`** overrides this setting.

```
Ntrace*Grid*foreground:
```

```
Ntrace*Grid*background:
Ntrace*Grid*font:
Ntrace*Grid*defaultDotsHigh:  30
Ntrace*Grid*defaultDotsWide:  60
```

The resource strings for the `File Selection Dialog Box` width, window titles, push buttons, and prompt strings follow. A `File Selection Dialog Box` is the type of window **`ntrace`** uses to prompt for file names, for example, configuration file names to open and save.

```
Ntrace*FileChooser*width:     180

Ntrace*OpenPopup*name:        Open Dialog
Ntrace*OpenPopup*open:        Open
Ntrace*OpenPopup*caption:     Enter configuration file name:
Ntrace*OpenPopup*geometry:

Ntrace*ReadPopup*name:        Read Dialog
Ntrace*ReadPopup*open:        Read
Ntrace*ReadPopup*caption:     Enter event-map file name:
Ntrace*ReadPopup*geometry:

Ntrace*SaveAsPopup*name:      Save As Dialog
Ntrace*SaveAsPopup*open:      Save
Ntrace*SaveAsPopup*caption:   Enter configuration file name to save:
Ntrace*SaveAsPopup*geometry:
```

The resource strings for the other dialog box titles and descriptive text are:

```
Ntrace*WarningDialog*name:           Warning Dialog

Ntrace*QuestionDialog*name:          Question Dialog

Ntrace*WorkingDialog*name:           Working Dialog

Ntrace*MacroDialog*name:             Macros
Ntrace*MacroDialog*caption:          List of Macros:

Ntrace*QualifiedEventDialog*name:    Qualified Events
Ntrace*QualifiedEventDialog*caption: List of Qualified Events:

Ntrace*QualifiedStateDialog*name:    Qualified States
Ntrace*QualifiedStateDialog*caption: List of Qualified States:
```

The resource strings for the window title and descriptive text for all Forms are:

```
Ntrace*SearchForm*name:                          Search

Ntrace*SummarizeForm*name:                       Summarize

Ntrace*SummarizeForm*TextScrollbox:defaultChars: 84
Ntrace*SummarizeForm*TextScrollbox:defaultLines: 14

Ntrace*SummarizeForm*SaveTextAsPopup*name: Save Summary Text As Dialog
Ntrace*SummarizeForm*SaveTextAsPopup*open: Save
Ntrace*SummarizeForm*SaveTextAsPopup*caption:
                                         Enter file name to save text
          to:
Ntrace*SummarizeForm*SaveTextAsPopup*geometry:
```

TIP:

If you sometimes work at a monochrome monitor, you may want to have two sets of the following X resource settings: one for color and one for monochrome. The color settings follow. The resource names for monochrome settings are identical except they say `Mono` instead of `Color`.

TIP:

Experiment with colors and shadings until you find a set you like. To avoid visual fatigue, use highly-contrasting colors and values sparingly.

The resource strings for the specific display objects are:

```
Ntrace*Color*GridLabel*background:
Ntrace*Color*GridLabel*foreground:
Ntrace*Color*GridLabel*font:
Ntrace*Color*GridLabel*textJustify:
Ntrace*Color*GridLabel*textGravity:

Ntrace*Color*DataBox*background:
Ntrace*Color*DataBox*foreground:
Ntrace*Color*DataBox*font:
Ntrace*Color*DataBox*textJustify:
Ntrace*Color*DataBox*textGravity:

Ntrace*Color*Column*background:
Ntrace*Color*Column*foreground:

Ntrace*Color*StateGraph*background:
Ntrace*Color*StateGraph*foreground:
Ntrace*Color*StateGraph*eventColor:

Ntrace*Color*EventGraph*background:
Ntrace*Color*EventGraph*foreground:

Ntrace*Color*DataGraph*background:
Ntrace*Color*DataGraph*foreground:

Ntrace*Color*Ruler*background:
Ntrace*Color*Ruler*foreground:
Ntrace*Color*Ruler*font:
Ntrace*Color*Ruler*markColor:
Ntrace*Color*Ruler*lostEventColor:
```

TIP:

On a monochrome display, make sure that you can differentiate among display objects within a Column. The easiest way to do this is to leave at least one grid dot between display objects in a Column and to make the background color of the Column black.

Grid object settings apply if you have not set the corresponding setting for a specific display object. The general grid object resource strings are:

```
Ntrace*Color*GridObject*background:
Ntrace*Color*GridObject*foreground:
Ntrace*Color*GridObject*borderColor:
```

For information about setting X resources for kernel displays, see “Color Information” on page 11-32.

Examples

Setting X resources to values is most consistent if the values of the X resources do not contain spaces. For example, even if your **`rgb.txt`** color file contains a color called “navy blue,” for simplicity type it as one word without any quotation marks.

In the following examples, you are making navy blue (`navyblue`) the foreground color (`foreground`) of all grid objects (`GridObject`) on a color monitor (`Color`) for **`ntrace`** (`Ntrace`). This example shows how this line may appear in your **`.Xdefaults`** file.

```
Ntrace*color*GridObject*foreground: navyblue
```

The following example shows how you can use this setting on the **`ntrace`** invocation line. Note: there must not be any spaces between the colon and the value.

```
$ ntrace -xrm Ntrace*color*GridObject*foreground:navyblue
```

Exercise: Customizing Display Colors

Edit your **`.Xdefaults`** file so it defines background colors for the following display objects. Suggested colors are provided.

Table B-2. Suggested Colors for X Resources

Display Object	Suggested Color
Column	CornflowerBlue
DataGraph	PowderBlue
StateGraph	LightSteelBlue
Ruler	PaleGreen
DataBox	Aquamarine
GridObject	SkyBlue

A possible solution follows:

```
Ntrace*Color*Column*background: CornflowerBlue
Ntrace*Color*DataGraph*background: PowderBlue
Ntrace*Color*StateGraph*background: LightSteelBlue
Ntrace*Color*Ruler*background: PaleGreen
Ntrace*Color*DataBox*background: Aquamarine
Ntrace*Color*GridObject*background: SkyBlue
```

To test your entries at an X terminal, invoke **`ntrace`** with the **`log`** trace event file, and bring up the default display page.

C
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_start()` call matches the one on the **`ntraceud`** invocation. Furthermore, check that the file exists and that you have permission to read and write it. Additionally, be sure your thread name contains no embedded spaces or punctuation, including periods. See "trace_start()" on page 3-5 and "trace_open_thread()" on page 3-9 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 3-11.

Q: How can I prevent user trace events from being discarded or lost?

A: Use expansive mode; avoid invoking **`ntraceud`** with the **`-filewrap`** and **`-bufferwrap`** options. Flush the shared memory buffer more often by tuning:

- The shared memory buffer size
- The shared memory buffer's flush percentage
- The **`ntraceud`** timeout interval

See "Preventing Trace Events Loss" on page A-1 and Chapter 4 for more information.

Q: What can I do if trace events are not appearing in an ntrace display?

A: Press `Refresh`, fill out the Search Form, fill in values in the interval control area, use the interval scroll bar, keep pressing the `Zoom Out` push button until you see trace events, examine a display object configuration so you know what it is "listening" for, add or reconfigure display objects on the grid. See Chapter 6 for more information.

Q: My trace event timings occasionally have huge gaps of time between them. What is the cause?

A: You are probably running your application on a Series 6000 system and are calling `clock_settime()`. This system call can corrupt the system interval timer which NightTrace uses for trace event timings.

Q: How can I get my kernel trace events to be mnemonically labeled when there is no vectors file or definition of the vector, syscall, and pid string tables anywhere?

A: Invoke **`ntfilter`** with the **`-v`** option. See “Converting KernelTrace Trace Event Files with ntfilter” on page 11-21.

Q: How can I prevent kernel trace events from being lost?

A:

- Verify that the raw kernel trace file is on a local file system and not an NFS file system.
- Ask your system administrator to increase the size of `TR_BUFFER_COUNT` kernel tunable parameter.

Glossary

This glossary defines terms used in the documentation. Terms in *italics* are defined here.

Ada task

An Ada task is a construct of statements which logically execute in parallel with other tasks within an Ada program (process). Tasks communicate asynchronously via variables whose visibility is defined by normal Ada scoping rules. Tasks communicate synchronously via rendezvous between a calling and accepting task.

Add

A *push button* that creates a new *macro*, *qualified event*, or *qualified state* on the current *display page*.

Apply

A *push button* that validates and saves all changes. The same functionality is available by pressing `<Enter>` in a modified field.

argument

See *trace event argument*.

boolean table

A pre-defined *string table* defined in the **`/usr/lib/NightTrace/tables`** file. It 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.

color display

An X server display that contains greater color variety than black, gray, and white. See also *monochrome display*.

Column

A *display object* that constrains the width of *StateGraphs*, *EventGraphs*, *Data-Graphs*, and *Rulers*.

configuration

The definition of a *display object*, *macro*, *qualified event*, or *qualified state*.

configuration file

An NightTrace-generated ASCII file that holds *display pages*, *macro*, *qualified event*, and *qualified state* definitions. This can also be a hand-edited table file, containing definition of *string tables* and/or *format tables*.

Configuration Form

The NightTrace form that allows you to define a *display object*'s data content, constraints, and graphic attributes, the value of a *macro* or the constraints of a *qualified event* or *qualified state*.

Configure

A *push button* that reconfigures and renames the selected *macro*, *qualified event*, or *qualified state*.

context switch

An action that occurs inside the kernel. Its functions are to save the state of the process that is currently executing, to initialize the state of the process to be run, and to begin execution of the new process.

context switch line

A vertical line superimposed on an *exception graph* or a *syscall graph* on a kernel *display page*. It indicates that the kernel has switched out the process that was previously running on the CPU and switched in a new process.

control

See *mouse button*, *push button* and *radio button*.

converted KernelTrace trace event file

A *KernelTrace trace event file* output by `ntfilter`. NightTrace reads it like any *trace event file*.

CPU box

A *GridLabel* 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*.

DataBox

A *display object* that displays possibly variable textual or numeric information.

DataGraph

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 *StateGraph* for each trace event logging process in your *trace event file(s)*.

Delete

Remove the selected *macro*, *qualified event*, or *qualified state*.

device table

A pre-defined, dynamically generated *string table* in the **`vectors`** file created by **`ntfilter`**. This 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: *GridLabels*, *DataBoxes*, *Columns*, *StateGraphs*, *EventGraphs*, *DataGraphs* and *Rulers*.

display page

The NightTrace *window* that allows you to layout *display objects* and see *trace event* and *state* information in them. You can store display pages in *configuration files*.

dotted area

See *grid*.

drag

To press and hold down a *mouse button* while moving the *mouse*. Usually you do this in NightTrace to position a *display object*.

duration

The period of time between the start and end *trace events* of some *state*.

Edit mode

The *display-page* mode that allows you to create, edit, and configure *display objects*, *macros*, *qualified events*, and *qualified states*. The other display-page mode is *View mode*.

ellipses (...)

An indicator at the end of a menu item that tells you this selection makes a *dialog box* appear. Also, an indicator in command line option summaries and syntax listings that tells you more than one occurrence of the previous syntactic component is allowed.

end function

A *state function* that provides information about the ending *trace event* of the *last completed instance of a state*. The *state* to which the end function applies is either the *qualified state* specified to the *function*, or the state being currently defined. Thus, if a 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* defined in **`/usr/lib/NightTrace/tables`**. It contains formats for statistical displays of trace event *matches* and type double *arguments.*

event_arg_summary table

A pre-defined *format table* defined in **`/usr/lib/NightTrace/tables`**. It contains formats for statistical displays of trace event *matches* and type long *arguments.*

EventGraph

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 *tags* or labels with numeric *trace event IDs*. The kernel's event-map file is **`/usr/lib/NightTrace/eventmap`**.

event_summary table

A pre-defined *format table* defined in **`/usr/lib/NightTrace/tables`**. It contains formats for statistical displays of trace event *matches* and trace event time *gaps*. It determines the default event-summary output format.

event table

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and maps all known numeric *trace event ID*s with symbolic *trace event tag*s.

event tag

See *trace event tag*.

exception

An event internal to the currently executing process that stops the current execution stream. Exceptions can be suspended and resumed.

exception graph

A *StateGraph* on a kernel *display page*. It displays *states* representing *exceptions* executing on the associated CPU.

expansive mode

The (default) mode that causes the **`ntraceud`** daemon to copy all *trace events* that ever reach the *shared memory buffer* to the indefinitely-sized *trace event file*. Invoke **`ntraceud`** without the **`-filewrap`** and **`-bufferwrap`** options to obtain this behavior. The two other **`ntraceud`** modes are *buffer-wraparound mode* and *file-wraparound mode*.

expression

A combination of operators and operands that evaluate to a value. Operands include constants, *macro* calls, *function* calls, *qualified events*, and *qualified states*.

Exit

A menu item that terminates an NightTrace session.

file-wraparound mode

The mode that causes the **`ntraceud`** daemon to overwrite the oldest *trace events* in the beginning of the *trace event file* with the newest ones; this means that **`ntraceud`** intentionally *discards* the oldest trace events to make room for the newest ones. Invoke **`ntraceud`** with the **`-filewrap`** option to obtain this behavior. The two other **`ntraceud`** modes are *expansive mode* and *buffer-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*.

GridLabel

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.

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 *DataGraph* 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 Time Start and Time End 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* and *KernelTrace* use to timestamp *trace events*.

KernelTrace

The tool that collects and textually analyzes system performance. It includes **`ktrace(1)`** and **`ntfilter(1)`**.

KernelTrace trace event file

A *trace event file* output by **`ktrace(1)`**. This can be used analyzed with **`ktrace`**, however it must be pre-processed by **`ntfilter`** before NightTrace can read it. See also *converted KernelTrace trace event file*.

keyboard

A traditional input device for entering text into fields. In this manual, this is a standard 101-key North American keyboard.

ktrace

A part of *KernelTrace* that is a stand-alone tool that can be used to extract system call (*syscall*), *exception*, *interrupt*, *context switch*, and device information from the kernel.

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 *DataBox* 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 *DataBox* 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 *DataBox* on a kernel *display page*. It displays the last *syscall* prior to the *current time line* that executed (and may still be executing) on the associated CPU.

lightweight process identifier

See *LWPID*.

lost trace event

A *trace event* **`ntraceud`** was unable to log. Several **`ntraceud`** options exist to prevent this trace event loss.

LWPID

An integer that represents an operating system lightweight process identifier. It makes up the second half of a *PID*.

macro

A user-defined named *expression* stored in a *configuration file*. When you call a macro, precede the macro name with a dollar sign.

mark

The solid triangle on a *Ruler* that points to a particular time.

match

A *trace event* or *state* that meets user-defined qualifying configuration criteria.

menu

A list of user-selectable choices.

menu bar

The horizontal band near the top of a *window* that contains a list of labeled *pull-down menus*.

message display area

The scrolling region of the *Global Window* or the *display page* that holds textual statistics, as well as error and warning messages.

monochrome display

A black, gray, and white X-server display. See also *color display.*

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, Fortran, or Ada; these applications may be composed of one or more processes, running on one or more CPUs. You can then examine these trace events and those from the kernel through the **`ntrace`** display utility.

NightTrace thread

A process, *thread* or *Ada task* (or a set of any combination of these) that is associated with a uniquely named *trace context*. The thread name is derived from the argument specified to the `trace_open_thread()` function.

NightTrace thread identifier

See *TID*.

NightView

A symbolic debugger that is part of the NightStar tool kit. It lets you debug C and Fortran applications; these applications may be composed of one or more processes, running on one or more CPUs. Among other things, NightView can automatically patch trace event logging routines into your executable application.

node

A system from which a *trace event file* can come from.

node box

If the RCIM synchronized tick clock is used to timestamp events, this is a *GridLabel* 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 created by **`ntfilter`**.

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.

ntfilter

A part of *KernelTrace* that converts a raw kernel trace file created by **`ktrace(1)`** into a *converted KernelTrace trace event file* that NightTrace can read.

ntrace display utility

The part of *NightTrace* that graphically displays *trace events*, trace event data, and *states* for debugging and performance analysis.

ntraceud

The *NightTrace* daemon process that allows you to log user-defined *trace events* and data from user applications written in C, Fortran, or Ada. These applications may be composed of one or more processes, running on one or more CPUs.

object

See *display object*.

offset

The number that identifies the position of a *trace event* in the chronologically-ordered sequence of trace events, regardless of the *trace event ID*. Counting starts from zero. For example, if a trace event with trace event ID 71 is the third trace event in the trace session, then its offset is 2.

offset function

A *function* that takes an *expression* that evaluates to an *offset* as a parameter.

OK

A *push button* that acknowledges the warning in a *dialog box*.

Open

A menu item and *push button* that opens an existing file.

ordinal trace event number

See *offset*.

panel

A *window* component that groups related buttons, for example *push buttons*.

PID

A 32-bit integer that represents an operating system process. The following syntax numerically specifies a PID: *raw_PID'LWPID.* The operating system process identifier (*raw PID*) is contained in the upper 16 bits and the lightweight process identifier (*LWPID*) is contained in the lower 16 bits.

PID table

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and associates process identifiers (*PIDs*) with process names. If kernel tracing, the `pid` string table in the **`vectors`** file created by **`ntfilter`** will be merged into this table.

point

To move the *mouse* so the mouse pointer is positioned at the place of interest.

pointer

A graphical symbol that represents the mouse pointer's current location in the *window*. The shape of the pointer shows the current usage. Usually a pointer is shaped like an arrow pointing to the upper left.

pop-up window

See *dialog box*.

press

To hold down a *mouse button* without releasing it or to depress a *keyboard* key.

pull-down menu

A list of related choices called menu items pulled down from the *menu bar*. *Click* on a menu item to select it.

push button

A graphic image of a labeled button. *Click* on a push button to select it.

qualified event

A user-defined **named** *event* configuration that consists of a set of one or more trace events, possibly restricted by an If-Expression, CPU List, TID List, PID List, and Node List. Qualified events provide a mechanism for referencing *trace events configurations* within certain *functions*. These definitions are stored in *configuration files*.

qualified state

A user-defined **named** *state* configuration that consists of a set of one or more states, possibly restricted by a Start-Expression, End-Expression, CPU List, TID List, PID List, and Node List. Qualified state provides a mechanism for referencing state *configurations* within certain *functions*. These definitions are stored in *configuration files*.

radio button

A graphic, labeled diamond-shape that represents a mutually exclusive selection from related radio buttons. *Click* on a radio button to select it.

raw PID

A 16-bit integer that makes up the first half of a *PID*.

RCIM

Real-Time Clocks and Interrupts Module. It provides a synchronized clock, edge-triggered interrupts, real-time clocks and programmable interrupts. Some set of interrupts can be distributed and sent to all connected RCIMs. The RCIM hardware is available via a standard PCI mezzanine card (PMC).

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 *DataBox* 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 DataBox 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.

shared memory buffer

The intermediate destination of *trace events* before **`ntraceud`** copies them to the *trace event file* on disk.

slider

The graphic part of a *scroll bar* that you move in the *trough* to change the display. This component is sometimes called a thumb.

spin lock

A device used to protect a resource, for example, the *shared memory buffer*.

start function

A *state function* that provides information about the start *event* of the *most recent instance of a state*. The *state* to which the start function applies is either the *qualified state* specified to the *function*, or the state being currently defined. Thus, if a qualfied 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 bounded by two trace events, a start *event* and an end *event*. An instance of a state is the period of time between the start event and end event, including the start and end events themselves. Additional conditions may be specified in a state definition to further constrain the state. Instances of states do not nest; that is, once a state becomes active, events that might normally satisfy the conditions for the start event are ignored until the end event is encountered. See also *qualified state*.

state function

The class of NightTrace *functions* which provide information about *states*, including: *start functions*, *end functions*, and *multi-state functions*.

StateGraph

A scrollable *display object* that graphically displays *states* as bars and *trace events* as vertical lines in a *Column*.

state_summary table

A pre-defined *format table* defined in **`/usr/lib/NightTrace/tables`**. It contains formats for statistical displays of state *matches*, state *durations*, and state time *gaps*. It determines the default state-summary output format.

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 *StateGraph* 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 created by **`ntfilter`**. 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

See *trace event tag*.

task

See *Ada task*.

task ID

A 16-bit integer chosen by the Ada run-time executive that uniquely identifies an *Ada task* within an Ada program.

text cursor

The blinking vertical bar in an editable text field that shows your current edit position within the field.

thread

A sequence of instructions and associated data that is scheduled and executed as an independent entity. Every UNIX process linked with the Threads Library contains at least one, and possibly many, threads. Threads within a process share the address space of the process.

thread ID

A 16-bit integer chosen by the threads library that uniquely identifies a *thread* within a given process.

TID

A 32-bit integer that represents a unique context to which *trace events* can be associated. The following syntax numerically specifies a TID: *raw_PID'task_id*, *raw_PID'thread_id*, or *raw_PID'0* (if neither *Ada tasks* nor *threads* are in use). The operating system process ID (*raw PID*) is contained in the upper 16 bits and either a *thread ID*, *task ID*, or zero is contained in the lower 16 bits.

TID table

A pre-defined, dynamically generated *string table*. It is internal to NightTrace and associates NightTrace thread identifiers (*TIDs*) with thread names. This table is not used in kernel tracing.

timestamp

The time at which a specific *trace event* was logged. This provides the means by which the chronology of the trace events logged by multiple processes can be assembled. The timestamp is obtained from the system *interval timer*, the *Time Base Register*, or the *RCIM synchronized tick clock*, depending on either the system architecture or user-specified options to **`ntraceud`**.

Time Base Register

The system timer on the Power Hawk/PowerStack systems that *NightTrace* and *KernelTrace* use to timestamp *trace events*.

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*). *KernelTrace* also records trace events.

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. See *converted KernelTrace trace event file*.

trace event function

The class of NightTrace *functions* that provide information about *trace events*. They operate on either the *qualified event* specified to that function or, if unspecified, the *current trace event*. Trace event functions include *multi-event functions*.

trace event ID

An integer that identifies a *trace event*. User trace event IDs are in the range `0-4095`, inclusive. Kernel trace event IDs are in the range `4100-4300`, inclusive.

trace event tag

A symbolic name mapped to a numeric *trace event ID* in an *event-map file.*

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 created by **`ntfilter`**. This string table contains the *interrupt* and *exception* vector names associated with the system on which the kernel tracing was performed.

View mode

The *display page* mode that allows you to see, search for, and summarize *trace event* information in the *message display area*, the *summary display area*, and *display objects* on the *grid*; create, edit, and configure *macros*, *qualified events*, and *qualified states*. The other display-page mode is *Edit mode.*

widget

A *window* component, for example a *scroll bar* or *push button*.

window

A rectangular screen area that permits the display and/or entry of data. The Night-Trace display utility consists of several windows.

window manager

The program that controls *window* placement, size, and operations.

wraparound mode

The mode that causes the **`ntraceud`** daemon to intentionally discard old events. There are two forms of wraparound mode: *buffer-wraparound* and *file-wrap-around*. The other **`ntraceud`** mode is *expansive mode.*

Index

Spine for 1.5” Binder

Product Name: 0.5" from top of spine, Helvetica, 36 pt, Bold

Volume Number (if any): Helvetica, 24 pt, Bold

Volume Name (if any): Helvetica, 18 pt, Bold

Manual Title(s): Helvetica, 10 pt, Bold, centered vertically within space above bar, double space between each title

Bar: 1" x 1/8" beginning 1/4" in from either side

Part Number: Helvetica, 6 pt, centered, 1/8" up

PowerMAX OS

User/Admin

NightTrace Manual

0890398