

QNX<sup>®</sup> Momentics<sup>®</sup> Development  
Suite

---

System Analysis Toolkit  
*User's Guide*

*For QNX<sup>®</sup> Neutrino<sup>®</sup> 6.2.0 or later*

© 2001 – 2005, QNX Software Systems. All rights reserved.

Printed under license by:

**QNX Software Systems Co.**  
175 Terence Matthews Crescent  
Kanata, Ontario  
K2M 1W8  
Canada  
Voice: +1 613 591-0931  
Fax: +1 613 591-3579  
Email: [info@qnx.com](mailto:info@qnx.com)  
Web: <http://www.qnx.com/>

Electronic edition published 2005

### **Technical support options**

To obtain technical support for any QNX product, visit the **Technical Support** section in the **Services** area on our website ([www.qnx.com](http://www.qnx.com)). You'll find a wide range of support options, including our free web-based **Developer Support Center**.

QNX, Momentics, Neutrino, and Photon microGUI are registered trademarks of QNX Software Systems in certain jurisdictions. All other trademarks and trade names belong to their respective owners.

# Contents

---

	<b>About This Guide</b>	<b>vii</b>
<b>1</b>	<b>Introduction</b>	<b>1</b>
	What is the System Analysis Toolkit (SAT)?	3
	Why is the SAT needed?	3
	How the SAT works	5
	The Instrumented Kernel	5
	Kernel buffer management	6
	The data-capture program	6
	Data interpretation	6
<b>2</b>	<b>What's in the SAT?</b>	<b>9</b>
<b>3</b>	<b>Events and the Kernel</b>	<b>13</b>
	What generates events	15
	Generating events: a typical scenario	15
	Multithreaded example:	15
	Thread context-switch time	17
	Thread restarting	18
<b>4</b>	<b>Kernel Buffer Management</b>	<b>19</b>
	Instrumented Kernel and kernel buffer management	21
	Buffer specifications	21
	Circular linked lists	22
	Linked list size	22

Full buffers and the high-water mark	22
Buffer overruns	23
<b>5 User Data Capture</b>	<b>25</b>
Overview	27
What the data-capture utility does	28
Tracing settings	29
Normal mode and daemon mode	29
Simple and combine events	29
Wide mode and fast mode	30
Tracebuffer file	30
Configuring	30
Configuring the Instrumented Kernel	30
Configuring data capture	31
<b>6 User Data Interpretation</b>	<b>33</b>
Overview	35
The <code>traceparser</code> library	36
Interpreting events	37
Simple and combine events	37
The <code>traceevent_t</code> structure	37
Event interlacing	38
Timestamp	38
The <code>traceprinter</code> utility	39
<b>7 Filtering</b>	<b>41</b>
Overview	43
Types of filters	43
Tradeoffs	44
The fast/wide mask	44
The static rules filter	45
The dynamic rules filter	46
The post-processing facility	47

## **8 Tutorial 49**

- Starting out 51
  - Setting up the Instrumented Kernel 51
  - Tutorial overview 51
- The `tracelogger` in normal mode 52
  - Interrupts, iterations, and events 52
  - Slow systems and busy systems 52
- The `tracelogger` in daemon mode 53
  - Overview 53
  - Gathering all events from all classes 54
  - Gathering all events from one class 55
  - Gathering five events from four classes 55
  - Gathering kernel calls 56
  - Event handling - simple 57
  - User event - simple 57

## **9 Hints and Troubleshooting 81**

- How to check for the Instrumented Kernel mode 83
- Run as `root` 83
- Monitor diskspace 83

## **10 Functions 85**

- `InterruptHookTrace()` 88
- `TraceEvent()` 90
- `traceparser()` 107
- `traceparser_cs()` 109
- `traceparser_cs_range()` 111
- `traceparser_debug()` 113
- `traceparser_destroy()` 115
- `traceparser_get_info()` 116
- `traceparser_init()` 120

## **11 Utilities 123**

tracelogger 126  
traceprinter 129

**A Kernel Call Arguments and Return  
Values 137**

**Index 159**

## ***About This Guide***

---



# ***Chapter 1***

---

## **Introduction**

### ***In this chapter...***

What is the System Analysis Toolkit (SAT)?	3
Why is the SAT needed?	3
How the SAT works	5



## What is the System Analysis Toolkit (SAT)?

In today's computing environments, developers need to monitor a dynamic execution of realtime systems with emphasis on their key architectural components. Such monitoring can reveal hidden hardware faults and design or implementation errors, as well as help improve overall system performance.

In order to accommodate those needs, we provide sophisticated tracing and profiling mechanisms, allowing execution monitoring in real time or offline. Because it works at the operating system level, the SAT, unlike debuggers, can monitor applications without having to modify them in any way.

The main goals for the SAT are:

- 1 Ease of use
- 2 Insight into system activity
- 3 High performance and efficiency with low overhead

## Why is the SAT needed?

In a running system, many things occur behind the scenes:

- Kernel calls are being made
- Messages are being passed
- Interrupts are being handled
- Threads are changing states — they're being created, blocking, running, restarting, and dying

The result of this activity are changes to the system state that are normally hidden from developers. The SAT is capable of intercepting these changes and logging them. Each event is logged with a *timestamp* and the *ID of the CPU* that handled it.

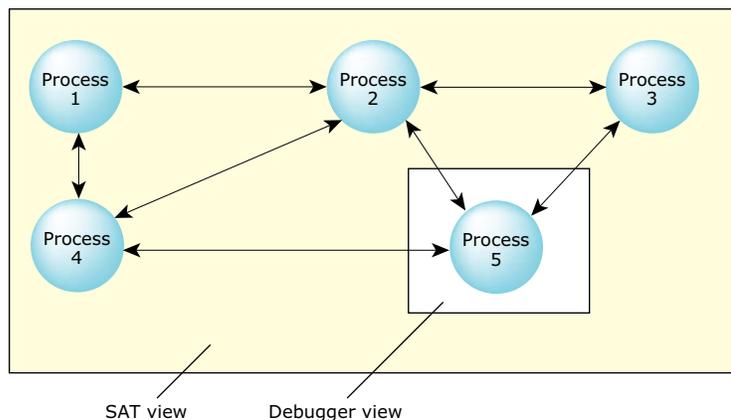


---

For a full understanding of how the kernel works, see the Neutrino Microkernel chapter in the *System Architecture* guide.

---

The SAT offers valuable information at all stages of a product's life cycle, from prototyping to optimization to in-service monitoring and field diagnostics.



---

*The SAT view and the debugger view*

In complicated systems, the information provided by standard debugging programs may not be detailed enough to solve the problem. Or, the problem may not be a bug as much as a process that's not behaving as expected. Unlike the SAT, debuggers lack the execution history essential to solving the many complex problems involved in "application tuning." In a large system, often consisting of many interconnected components or processes, traditional debugging, which lets you look at only a single module, can't easily assist if the problem lies in how the modules interact with each other. Where a debugger can view a single process, the SAT can view *all* processes at the same time. Also, unlike debugging, the SAT doesn't need code augmentation and can be used to track the impact of external, precompiled code.

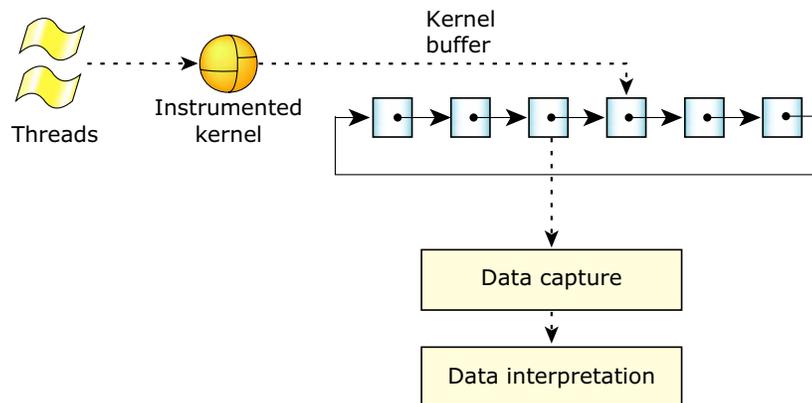
Because it offers a system-level view of the internal workings of the kernel, the SAT can be used for *performance analysis* and *optimization* of large interconnected systems as well as single processes.

It allows *realtime debugging* to help *pinpoint deadlock and race conditions* by showing what circumstances led up to the problem. Rather than just a “snapshot”, the SAT offers a “movie” of what’s happening in your system.

Because the instrumented version of the kernel runs with negligible performance penalties, you can optionally leave it in the final embedded system. Should any problems arise *in the field*, you can use the SAT for *low-level diagnostics*.

The SAT offers a nonintrusive method of instrumenting the code — programs can literally monitor themselves. In addition to *passive/non-intrusive event tracing*, you can *proactively trace events* by injecting your own “flag” events.

## How the SAT works



## The Instrumented Kernel

The Instrumented Kernel is actually the regular QNX microkernel with a small, highly efficient event-gathering module included. Except for the instrumentation, its operation is virtually

indistinguishable — the Instrumented Kernel runs at 98% of the speed of our regular microkernel.

As threads run, the Instrumented Kernel continuously intercepts information about what the kernel is doing, generating time-stamped and CPU-stamped events that are stored in a *circular linked list of buffers*. Because the tracing occurs at the kernel level, the SAT can track the performance of *all* processes, including the data-capturing program.

## Kernel buffer management

The kernel buffer is composed of many small buffers. Although the number of buffers is limited only by the amount of system memory, it's important to understand that this space must be managed carefully. If *all* of the events are being traced on an active system, the number of events can be quite large.

To allow the Instrumented Kernel to write to one part of the kernel buffer and store another part of it simultaneously, the kernel buffer is organized as a circular linked list. As the buffer data reaches a high-water mark (about 70% full), the Instrumented Kernel module sends a signal to the data-capture program with the address of the buffer. The data-capture program can then retrieve the buffer and save it to a storage location for offline processing or pass it to a data interpreter for realtime manipulation. In either case, once the buffer has been “emptied,” it is once again available for use by the kernel.

## The data-capture program

The SAT includes a **tracelogger** you can use to capture data. This data-capture program outputs the captured data in raw binary format to a device or file for processing.

## Data interpretation

To aid in processing the binary trace event data, we provide the **libtraceparser** library. The API functions let you set up a series of functions that are called when complete buffer slots of event data have been received/read from the raw binary event stream.

We also provide a linear trace event printer (**traceprinter**) that outputs all of the trace events ordered linearly by their timestamp as they are emitted by the kernel. This utility makes use of the **libtraceparser** library. Advanced users may wish to either customize **traceprinter** to make their own output program or use the API to create an interface to do the following *offline* or in *real time*:

- perform analysis
- display results
- debug applications
- system self-monitoring
- show events ordered by process or by thread
- show thread states and transitions
- show currently running threads.



***Chapter 2***

---

**What's in the SAT?**



The QNX System Analysis Toolkit (SAT) consists of the following main components:

#### Instrumented Kernel

The Instrumented Kernel is an integral part of the SAT. In addition to being a fully functional QNX RTOS microkernel, it also intercepts time events and passes them to the **tracelogger** daemon for further processing.

#### The **tracelogger** utility

The **tracelogger** daemon receives events from the Instrumented Kernel and saves them to a file/device for later analysis.

#### The **traceprinter** utility

The **traceprinter** utility displays the trace event file in a readable form. Users can also modify the **traceprinter** source as a basis for their own custom programs.

#### Trace Control Interface

The SAT trace API uses a single *TraceEvent()* kernel call. The *TraceEvent()* call has over 30 different execution modes that enable the user to:

- create internal trace buffers
- set up filters
- control the tracing process
- insert user defined events

#### The Traceparser Library

The Traceparser library provides an API for parsing and interpreting the trace events that are stored in the event file. The library simplifies the parsing and interpretation process by allowing the user to easily:

- set up callback functions and associations for each event
- retrieve header and system information from the trace event file
- debug and control the parsing process.

**Examples** You'll find numerous sample code listings (and source) that will take you from gathering a simple set of events through to sophisticated filtering and inserting your own events. You can modify the source and use snippets of it to more quickly create your own applications.

**Source code** You have full source code for all of the following:

- the **tracelogger** utility
- the **traceprinter** utility
- the **traceparser** library
- sample test programs
- tutorial example programs

## ***Chapter 3***

---

# **Events and the Kernel**

### ***In this chapter...***

What generates events	15
Generating events: a typical scenario	15



## What generates events

The QNX microkernel generates events for more than just system calls. The following are some of the activities that generate events:

- kernel calls
- scheduling activity
- interrupt handling
- thread/process creation, destruction, and state changes.

In addition, the Instrumented Kernel also inserts “artificial” events for:

- time events
- user events that may be used as “marker flags.”

Also, single kernel calls or system activities may actually generate more than one event.

## Generating events: a typical scenario

One of the powerful features of the QNX RTOS is its ability to run multiple threads. Having more than one thread increases the level of complexity — the OS must handle threads of differing priorities competing with each other.

### Multithreaded example:

In our example we’ll use two threads:

<b>Thread</b>	<b>Priority</b>
A	High
B	Low

Now we’ll watch them run, assuming both “start” at the same time:



When logging starts, the Instrumented Kernel sends information about each thread. Existing processes will appear to be created during this procedure.

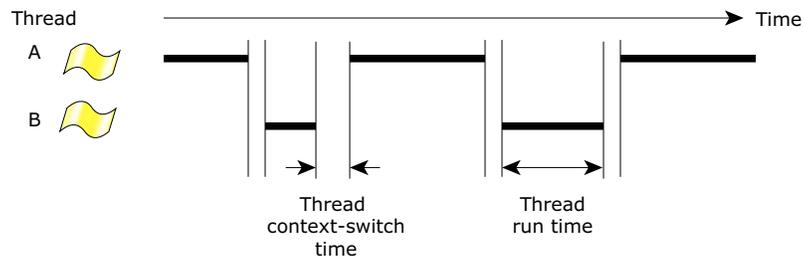
Time	Thread	Action	Explanation
t1	A	Create	Thread is created.
t2	A	Block	The thread is waiting for, say, I/O; it can't continue without it.
t3	B	Create	Rather than sit idle, the kernel runs next highest priority thread.
t4	B	Kernel Call	Thread B is working.
t4.5	n/a	n/a	I/O completed; Thread A is ready to run.
t5	B	Block	Thread A is now ready to run — it preempts thread B.
t6	A	Run	Thread A resumes.
t7	A	Dies	Its task complete, the thread terminates.
t8	B	Runs	Thread B continues from where it left off.

*continued...*

Time	Thread	Action	Explanation
t9	...and so on...	...and so on...	...and so on...

## Thread context-switch time

Threads don't switch instantaneously — after one thread blocks or yields to another, the kernel must save the settings before running another thread. The time to save this state and restore another is known as *thread context-switch time*. This context-switch time between threads is small, but important.



### *Thread context switching*

In some cases, two or more threads (or processes) may switch back and forth without actually accomplishing much. This is akin to two overly polite people each offering to let the other pass through a narrow door first — neither of them gets to where they're going on time (two aggressive people encounter a similar problem). This type of problem is exactly what the SAT can quickly and easily highlight. By showing the context-switch operations in conjunction with thread state transitions, you can quickly see why otherwise fast systems seem to “crawl.”

## Thread restarting

In order to achieve maximum responsiveness, much of the QNX microkernel is fully preemptable. In some cases, this means that when a thread is interrupted in a kernel call, it won't be able to restart exactly where it began. Instead, the kernel call will be restarted — it “rewinds” itself. The SAT tries to hide the spurious calls but may not succeed in suppressing them all. As a result, it's possible to see several events generated from a specific thread that has been preempted. If this occurs, the last event is the actual one.

## ***Chapter 4***

---

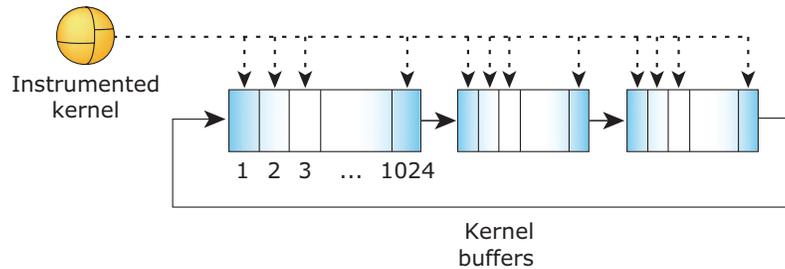
# **Kernel Buffer Management**

### ***In this chapter...***

Instrumented Kernel and kernel buffer management	21
Buffer specifications	21
Circular linked lists	22



## Instrumented Kernel and kernel buffer management



### *The kernel buffers*

As the Instrumented Kernel intercepts events, it stores them in a *circular linked list of buffers*. As each buffer fills, the Instrumented Kernel sends a signal to the data-capturing program that the buffer is ready to be read.

## Buffer specifications

Each buffer is of a fixed size and is divided into a fixed number of slots:

Event buffer slots per buffer	1024
Event buffer slot size	16 bytes
Buffer size	16 K

Some events are *single buffer slot events* (“simple events”) while others are *multiple buffer slot events* (“combine events”). In either case there is only *one* event, but the number of *event buffer slots* required to describe it may vary.

For details, see the User Data Interpretation chapter.

## Circular linked lists

### Linked list size

Although the size of the buffers is fixed, the maximum number of buffers used by a system is limited only by the amount of memory. (The `tracelogger` utility uses a default setting of 32 buffers, or about 500 K of memory.)

The buffers share kernel memory with the application(s) and the kernel automatically allocates memory at the request of the data-capture utility. The kernel allocates the buffers *contiguous physical memory space*. If the data-capture program requests a larger block than is available contiguously, the Instrumented Kernel will return an error message.

For all intents and purposes, the number of events the Instrumented Kernel generates is infinite. Except for severe filtering or logging for only a few seconds, the Instrumented Kernel will probably exhaust the circular linked list of buffers, no matter how large it is. To allow the Instrumented Kernel to continue logging indefinitely, the data-capture program must continuously pipe (empty) the buffers.

### Full buffers and the high-water mark

As each buffer becomes *full* (more on that shortly), the Instrumented Kernel sends a signal to the data-capturing program to save the buffer. Because the buffer size is fixed, the kernel sends only the buffer address; the length is constant.

The Instrumented Kernel can't flush a buffer or change buffers within an interrupt. If the interrupt wasn't handled before the buffer became 100% full, some of the events may be lost. To ensure this never happens, the Instrumented Kernel requests a buffer flush at the *high-water mark*.

The high-water mark is set at an efficient, yet conservative, level of about 70%. Most interrupt routines require fewer than 300 event buffer slots (approximately 30% of 1024 event buffer slots), so there's virtually no chance that any events will be lost. (The few routines that

use extremely long interrupts should include a manual buffer-flush request in their code.)

Therefore, in a normal system, the kernel logs about 715 events of the fixed maximum of 1024 events before notifying the capture program.

## Buffer overruns

The Instrumented Kernel is both the very core of the system and the controller of the event buffers.

When the Instrumented Kernel is busy, it logs more events. The buffers fill more quickly and the Instrumented Kernel requests buffer-flushes more often. The data-capture program handles each buffer-flush request; the Instrumented Kernel switches to the next buffer and continues logging events. In an extremely busy system, the data-capture program may not be able to flush the buffers as quickly as the Instrumented Kernel fills them.

In a three-buffer scenario, the Instrumented Kernel fills **Buffer 1** and signals the data-capture program that the buffer is full. The data-capture program takes “ownership” of **Buffer 1** and the Instrumented Kernel marks the buffer as “busy/in use.” If, say, the file is being saved to a hard drive that happens to be busy, then the Instrumented Kernel may fill **Buffer 2** and **Buffer 3** before the data-capture program can release **Buffer 1**. In this case, the Instrumented Kernel skips **Buffer 1** and writes to **Buffer 2**. The previous contents of **Buffer 2** are overwritten and the timestamps on the event buffer slots will show a discontinuity.

For more on buffer overruns, see the Tutorial chapter.



## **Chapter 5**

---

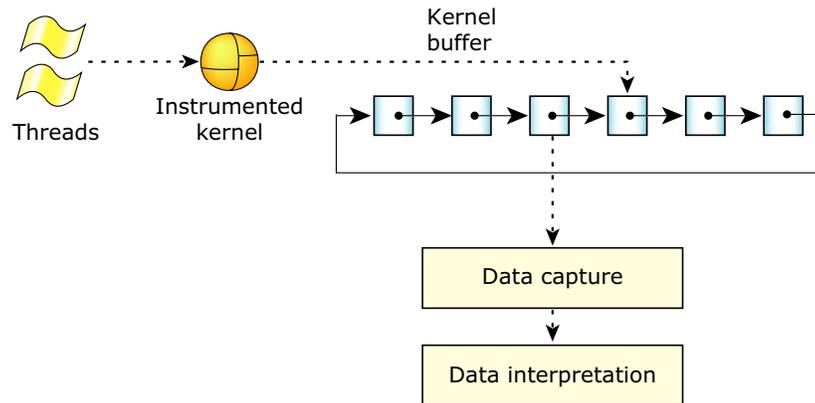
# **User Data Capture**

### ***In this chapter...***

Overview	27
What the data-capture utility does	28
Tracing settings	29
Configuring	30



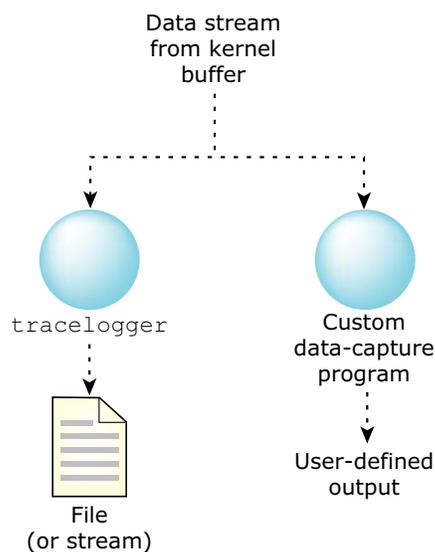
## Overview



### *Overall view of the SAT*

We've provided **tracelogger** as the default data-capture program. Although it's possible for you to write your own utility from scratch, there's little need to. More likely, you would tailor the **tracelogger** code to suit your own needs. Although this section describes the data-capturing process generically, much of it will focus on **tracelogger**'s capabilities.

## What the data-capture utility does



---

### *Possible data capture configurations*

Some of the general concepts are explained here, but for a full description of the utility and its options, see **tracelogger** in the Utilities chapter in this guide.

The data-capture utility is the “messenger” between the Instrumented Kernel and the filesystem.

The main function of the data-capture utility is to save the buffers given to it by the Instrumented Kernel to an output device (which may be a file or something else). In order to accomplish this function, the utility must also:

- interface with the Instrumented Kernel
- specify data-filtering requirements the Instrumented Kernel will use.

The Instrumented Kernel is fully configurable to log *all* events or only a specified few. Also, it can gather simplified information (in “fast” mode) or detailed information (in “wide” mode).

## Tracing settings

### Normal mode and daemon mode

The `tracelogger` can run in one of two modes. Below is an outline of the strengths, weaknesses, and features of both:

Feature	Normal mode	Daemon mode
<b>tracelogger support</b>	Full	Limited
<b>Controlability</b>	Limited	Full
<b>Events recorded by default</b>	All	None
<b>Configuration difficulty</b>	Easy	Harder
<b>Configuration method</b>	Command line only	User program, using calls to <code>TraceEvent()</code>
<b>Logging starts</b>	Instantaneously	Through a user program; also calls to <code>TraceEvent()</code>

### Simple and combine events

Most events can be described in a *single event buffer slot* (“simple event”). When there’s too much information to describe the event in a single buffer slot, the event is described in *multiple event buffer slots* (“combine event”). These events buffer slots all look the same so there’s no need for the data-capture program to distinguish between them.

For more information about simple events and combine events, see the User Data Interpretation chapter.

## Wide mode and fast mode

In *wide mode*, the Instrumented Kernel uses as many buffer slots as are necessary to fully log the event — the amount of space is theoretically unlimited and can span several kilobytes for a single event. Except for rare occasions, it doesn't exceed *four* 16-byte spaces.

In *fast mode*, the Instrumented Kernel uses *only one* buffer slot per event, no matter how many event buffer slots would be used to describe it in wide mode. Because *simple events* require only a *single event buffer slot* anyway, they are logged completely. On the other hand, *combine events* are “incomplete” because only the first of the event buffer slots is logged for each event.

For a detailed list of events and their respective entries, see the appendix.

## Tracebuffer file

Because the circular linked list of buffers can't hope to store a complete log of event activity for any significant amount of time, the tracebuffer must be handed off to a data-capture program. Normally the data-capture program would pipe the information to either an output device or a file.

By default, the `tracelogger` utility saves the output to the binary file `/dev/shmem/tracebuffer` but you can specify a filename.

# Configuring

## Configuring the Instrumented Kernel

You must configure the Instrumented Kernel before logging. The Instrumented Kernel configuration settings include:

- buffer allocations (size)
- which events and classes of events are logged (filtering)
- whether to log in wide mode or fast mode.

The event configurations are made in an additive or subtractive manner — starting from no events, specific classes or events may be added, or starting from all events, specific ones may be excluded.

You may make these configurations through the data-capture program or another, independent program. The Instrumented Kernel retains the settings.



---

Multiple programs access a single Instrumented Kernel configuration. Changing settings in one process supercedes the settings in another. For this reason, you should always make the configurations in an additive manner when using multiple processes.

---

## Configuring data capture

You must perform the data-capture configuration using a custom program. The **tracelogger** utility will do only very basic configuring, and only in normal mode, where it logs *all* events.

The custom program:

- manually starts and stops logging
- manually sends buffer flush requests (if needed)
- saves the buffer (to a file or other output method)
- configures all filters except the post-processing filters (the filters control which events and classes of events are logged.)

Note that the custom program would be created by the user.

The **tracelogger** utility handles only the buffer-saving function.

For configuration details, see the *TraceEvent()* function in the Functions chapter.

For sample configuration files, see the Tutorial chapter.



## **Chapter 6**

---

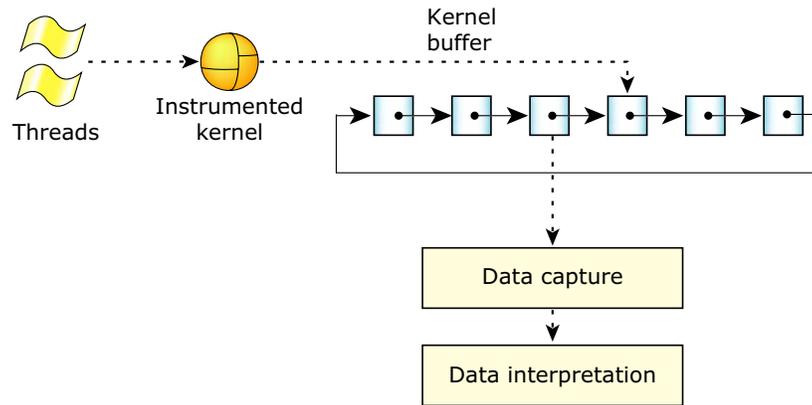
# **User Data Interpretation**

### ***In this chapter...***

Overview	35
Interpreting events	37
The <code>traceprinter</code> utility	39

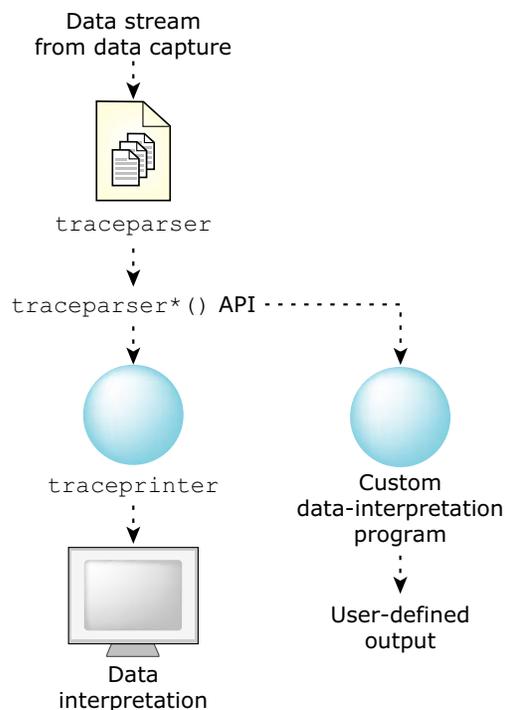


## Overview



*Overall view of the SAT*

Once the data has been captured, you may process it, either in real time or offline.



*Possible data interpretation configurations*

## The traceparser library

The **traceparser** library provides a front end to facilitate the handling and parsing of events received from the Instrumented Kernel and the data-capture utility. The library serves as a thin middle layer to:

- assemble multiple buffer slots into a single event
- perform data parsing to execute user-defined callbacks triggered by certain events.

## Interpreting events

### Simple and combine events

A simple event is an event that can be described in a *single event buffer slot*; a combine event is an event that is larger and can be fully described only in *multiple event buffer slots*. Both simple and combine events consist of only *one* kernel event.

Each event buffer slot is an opaque `traceevent_t` structure.

### The `traceevent_t` structure



The `traceevent_t` structure is *opaque* — although some details are provided, the structure shouldn't be accessed without the `libtraceparser` API.

The `traceevent_t` structure is only 16 bytes long, and only half of that describes the event. This small size reduces instrumentation overhead and improves granularity. Where the information required to represent the event won't fit into a single `traceevent_t` structure, it spans as many `traceevent_t` structures as required, resulting in a *combine event*. A combine event isn't actually several events combined, but rather a single, long event requiring a combination of `traceevent_t` elements to represent it.

In order to distinguish regular events from combine events, the `traceevent_t` structure includes a 2-bit flag that indicates whether the event is a single event or whether it's the first, middle, or last `traceevent_t` structure of the event. The flag is also used as a rudimentary integrity check. The timestamp element of the combine event is identical in each buffer slot; no other event will have the same timestamp.

Adding this “thin” protocol doesn't burden the Instrumented Kernel and keeps the `traceevent_t` structure small. The tradeoff, though, is that it may take many `traceevent_t` structures to represent a single kernel event.

## Event interlacing

Although events are timestamped immediately, they may not be written to the buffer in one single operation (atomically). When *multiple buffer slot events* (“combine events”) are being written to the buffer, the process is frequently interrupted in order to allow other threads and processes to run. Events triggered by higher-priority threads are often written to the buffer first. Thus, events may be *interlaced*. Although events may not be contiguous, they are *not* scrambled (unless there’s a buffer overrun.) The sequential order of the combine event is always correct, even if it’s interrupted with a different event.

In order to maintain speed during runtime, the Instrumented Kernel writes events unsorted as quickly as possible; reassembling the combine events must be done in post-processing. The `libtraceparser` API transparently reassembles the events.

## Timestamp

The timestamp is the 32 Least Significant Bits (LSB) part of the 64-bit clock. Whenever the 32-bit portion of the clock rolls over, a control event is issued. Although adjacent events will never have the same exact timestamp, there may be some timestamp duplication due to the clock’s rolling over.

The rollover control event includes the 32 Most Significant Bits (MSB), so you may reassemble the full clock time, if required. The timestamp includes only the LSB in order to reduce the amount of data being generated. (A 1 GHz clock rolls over every 4.29 sec — an eternity compared to the number of events generated.)



---

Although the majority of events are stored chronologically, you shouldn't write code that depends on events being retrieved chronologically. Some *multiple buffer slot events* (combine events) may be interlaced with others with *leading* timestamps. In the case of buffer overruns, the timestamps will definitely be scrambled, with entire blocks of events out of chronological order. Spurious gaps, while theoretically possible, are very unlikely.

---

## The **traceprinter** utility

The **traceprinter** utility consists of a long list of callback definitions, followed by a fairly simple parsing procedure. Each of the callback definitions is for printing.

In its native form, the **traceprinter** utility is of limited use. However, it's been designed to be easily borrowed from, copied, or modified to allow you to customize your own utilities. See the copyright header included in the source for the full terms of use.

For functionality details, see the **traceprinter** utility in the Utilities chapter in this Guide.



## **Chapter 7**

---

# **Filtering**

### ***In this chapter...***

Overview	43
The fast/wide mask	44
The static rules filter	45
The dynamic rules filter	46
The post-processing facility	47



## Overview

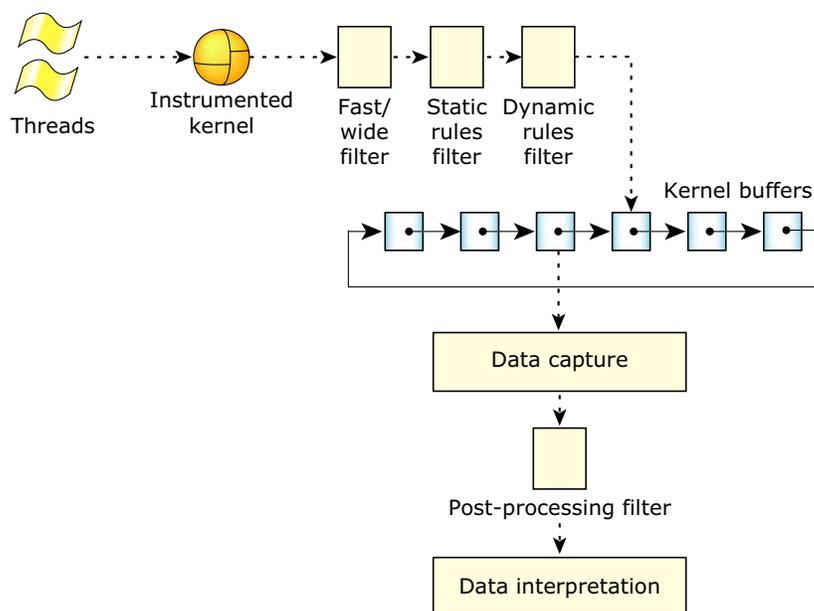
Because the amount of data the Instrumented Kernel generates can be overwhelming, the SAT includes several filters to reduce the amount of data to be processed. The filters can be customized to perform complex decision-making before filtering and other tasks.

### Types of filters

The following filters are available:

- Fast/wide mask
- Static rules filter
- Dynamic rules filter
- Post-processing facility

Except for the post-processing facility, all the filters affect the amount of data being logged into the kernel buffers. Unlike the other filters, the post-processing facility doesn't discard data; it simply doesn't use it. If the data is stored, it can always be used later.



Overall view of the SAT and its filters

## Tradeoffs

There are tradeoffs to consider when filtering:

- Gathering many events generates a lot of data, which requires *memory* and *processor time*.
- Filtered data is discarded; there's a chance some of the filtered data was useful (this doesn't apply to the post-processing facility.)

## The fast/wide mask

The fast/wide mask affects whether data is gathered in fast mode or wide mode. The fast/wide settings can be made on a per-event class and type basis — some can be fast while others are wide.

In wide mode, the entire event is stored, using as many buffer slots as required. In fast mode, each event is stored as a single buffer slot —

“multiple buffer slot events” (combine events) are simplified down to a single buffer slot.

In general, wide mode will generate several times more data than fast mode.




---

The fast/wide filter does *not* clip the tail end of “multiple buffer slot events” (combine events); it summarizes the most important aspects of the event in a single buffer slot. Thus, the first element of a “multiple buffer slot event” (combine event) *may not* be the same in fast and wide mode. (The fast/wide filter isn’t a mask.)

---

The filter is set using the *TraceEvent()* function:

```
TraceEvent (
    _NTO_TRACE_SETALLCLASSESWIDE ); /* wide mode */
TraceEvent (
    _NTO_TRACE_SETALLCLASSESFAST ); /* fast mode */
```

For an example of using fast or wide mode, see the `all_classes.c` example in the Tutorial chapter.

For the specific output differences between fast and wide mode, see the Kernel Call Arguments and Return Values appendix.

## The static rules filter

The static rules filter uses pre-defined classes and types defined in *TraceEvent()* that may be modified “on the fly” using a custom program.

The static rules filter is the best, most efficient method of data reduction. Because it’s a fast filter, severe filtering will generally free up the processor while reducing the data rate to a comparative trickle. The filter is also useful for gathering large amounts of data periodically, or after many hours of logging without generating gigabytes of data in the interim.

Set the filter using the *TraceEvent()* function:

```
TraceEvent (
    _NTO_TRACE_ADDCLASS,
    _NTO_TRACE_INTENTER );
TraceEvent (
    _NTO_TRACE_SETCLASSPID,
    _NTO_TRACE_KERCALL,
    pid ); /* "pid" is pid number */
TraceEvent (
    _NTO_TRACE_SETCLASSTID,
    _NTO_TRACEPROCESS,
    pid,
    tid ); /* "tid" is tid number */
TraceEvent (
    _NTO_TRACE_ADDEVENT,
    _NTO_TRACE_THREAD,
    _NTO_TRACE_THRUNNING );
```

For an example using the static filter, see the `five_events.c` example in the Tutorial chapter.

## The dynamic rules filter

The dynamic rules filter can do all the filtering that the static filter does, and more, but it isn't as fast.

The dynamic rules filter can be “turned on” with `_NTO.TRACE.ADDEVENTHANDLER` and `_NTO.TRACE.ADDCLASSEVHANDLER` modes. Using these modes, the user can attach its own custom defined function (event handler) that will control emission of trace events.

The dynamic filter is an event handler that works like an interrupt handler. When this filter is used, a section of your custom code is executed. The code can test for a set of conditions before determining whether the event should be stored. The code must return 0 to *prevent* the event from being stored. On any return except zero, the event is stored.

For example, a dynamic control of one particular event from one particular class can be performed using `_NTO.TRACE.ADDEVENTHANDLER` interface mode as:

```
TraceEvent(_NTO_TRACE_ADDEVENTHANDLER,  
           class,  
           event,  
           int (*event_hdlr)(event_data_t*),  
           event_data_t* data_struct)
```

A permanent “ON” state of the filter can be achieved with a very simple event handler defined as:

```
int event_handler(event_data_t* dummy_pt)  
{  
    return(1);  
}
```

In addition to deciding whether or not the event should be logged, the dynamic rules filter can be used to output events to external hardware or to perform other tasks — it’s up to you because it’s your code. Naturally, you should write the code as efficiently as possible in order to minimize the overhead.

It is possible to access the information about the intercepted event within the event handler. This can be done by examining members of `event_data_t` structure passed as an argument to the event handler. More detail description of the `event_data_t` members can be found in **Dynamic rules filter configuration** of the `TraceEvent()` function.

For example of using the dynamic filter, see the `eh_simple.c` example in the Tutorial chapter.

## The post-processing facility

The post-processing facility is different from the other filters in that it reacts to the events without permanently discarding them (or having to choose not to). Because the processing would be done on the captured data, often saved as a file, you could make multiple passes on the same data without changing it — one pass could count the number of thread state changes, another pass could display all the kernel events.

The post-processing facility is really a collection of callback functions that decide what to do for each event.

For an example of a post-processing facility, see the source code for **traceprinter**.

## Chapter 8

---

### Tutorial

#### *In this chapter...*

Starting out	51
The <code>tracelogger</code> in normal mode	52
The <code>tracelogger</code> in daemon mode	53



## Starting out

### Setting up the Instrumented Kernel

The Instrumented Kernel is a drop-in replacement for the noninstrumented kernel. In fact, when you're not tracing any events, the Instrumented Kernel operates at almost the same speed as the normal kernel. To use the Instrumented Kernel rather than the normal kernel:

- 1 In your buildfile, replace the entry for `procnto` with `procnto-instr`.
- 2 Run the `mkifs` utility to rebuild the image.
- 3 Replace your current boot image with the new one.
- 4 Add `tracelogger` and `traceprinter` to your buildfile or target.

Note that the Instrumented Kernel is slightly larger than the normal kernel.

For more information about `mkifs`, see the *Utilities Reference*.

### Tutorial overview

As you go through this tutorial, we'll ask you to save files.

The "tracebuffer" files are binary files that can be interpreted only with the `libtraceparser` library, which the `traceprinter` utility uses. To help you ensure you're getting the right results, we've used the same names in our documentation. For example, when we ask you to save to the file `tb.bare`, we'll show the `traceprinter` output here as `tb.bare.txt`, which is a file you can view with your favorite text editor.

Because the default number of buffers is 32, which produces a rather large file, we'll use the `-n` option to limit the number of buffers to a reasonable number. Feel free to use the default, but expect a large file.

## The `tracelogger` in normal mode

### Interrupts, iterations, and events

With no other processes running, type: `tracelogger -f tb.bare`

You'll see that the `interrupts` increment concurrently with the `iterations`. This is a good indication that the system was partially idle during the logging process.

The interrupts are not the traditional interrupts — they're "fake" ones during which the `tracelogger` saves the buffer just before flushing it.

You should also note that every entry for `events` tends to be 715, or very nearly 715. This is a count of how many events were logged in each buffer before `tracelogger` issued an interrupt to start writing to a new buffer. The default buffer holds 1024 events, numbered from 0 to 1023. At the high-water mark, when the buffer is 70% full, the system writes the buffer to file and flushes the buffer from memory.

### Slow systems and busy systems

The Instrumented Kernel kernel has been designed to be quite rugged and to have only a small performance penalty. Because the `tracelogger`, in its normal mode, saves large chunks of data, the amount of data it has to handle may overwhelm slow systems or systems with limited resources.

Since we all have different systems and there's no easy way to slow them down consistently, we'll cheat and set the buffer to 1. The default number of buffers is 32, which is plenty to keep the system running along without problems. When the buffer is only 1, the system tries to save the buffer at the high-water mark while at the same time rewriting it from the beginning. Because it's impossible to save the entire buffer before any more kernel calls occur, what's saved is garbage data or, at best, partial data.

Enter:

```
tracelogger -b 1
```

The results show the number of events as being nowhere near 715. Even though it won't happen in this example, *if the number of events is 0, you likely have a problem.*

As a general rule, the number of events should be consistently the high-water mark of the maximum number of events.

The **iterations** are the number of times the system has written to the buffer and the **interrupts** are an indication of how often the system saved (or tried to save) the buffer. If the number of **interrupts** is less than the number of **iterations**, it's a good indication that some data has been lost. This is particularly true if the number of **events** is ever shown to be 0. It's not serious if **interrupts** trail **iterations** by only 1.

## The **tracelogger** in daemon mode

### Overview

#### Normal mode vs. Daemon mode

In normal mode, **tracelogger** by default gathers *all* events in *fast* mode.

In daemon mode, **tracelogger** gathers *no* events — you must specify the events you wish to gather.

When you run **tracelogger** from the command line, the default is in normal mode. Normal mode allows for fast setup but the amount of information it generates is overwhelming.

Daemon mode can provide only the information you want, but at the expense of setup time and complexity. To start daemon mode, use the **tracelogger -d1** option. In daemon mode, **tracelogger** waits for a signal to start. We'll provide the start signal with a program.

#### Understanding the tutorial format

To reduce repetition and keep the programs simple, some functionality has been put into a header file. You'll have to save the included **instrex.h** file in order for the programs to compile.

Each of these sample programs uses a similar format:

- 1 Compile the `C file` into a file of the same name, without the `.c` extension
- 2 Run the `Command`.
- 3 In a *separate* terminal window, run the compiled `C file`. Some examples use options. Watch the first terminal window. In a few seconds the `tracelogger` will finish running.
- 4 If you run the program, it generates its own `sample result file`; if you didn't run the program, take a look at our `traceprinter` output. (Note that different versions and systems will create slightly different results.)



---

You may include these samples in your code as long as you comply with the copyright.

---

## Gathering all events from all classes

C File            `all_classes.c`

Command        `tracelogger -d1 -n 3 -f`  
                 `all_classes.results`

Running C File  
                 `all_classes`

Sample result file  
                 `all_classes.results.txt`

Viewing        `traceprinter -f all_classes.results`

In our first example, we'll set up daemon mode to gather *all* events from *all* classes. Despite how quickly the program ran, the amount of data it generated is rather overwhelming. Feel free to interrupt the `traceprinter` — it'll list data for a while.

This example demonstrates the capability of the trace module to capture *huge* amounts of data about the events.

While it's good to know how to gather everything, we'll clearly need to be able to refine our search.

## Gathering all events from one class

C File            **one\_class.c**

Command        **tracelogger -d1 -n 3 -f  
                 one\_class.results**

Running C File

**one\_class**

Sample result file

**one\_class.results.txt**

Viewing        **traceprinter -f one\_class.results**

Now we'll gather *all* events from only *one* class — `_NTO_TRACE_THREAD`. This class is arbitrarily chosen to demonstrate filtering by classes; there's nothing particularly special about this class versus any other. For a full list of the possible classes, see *TraceEvent()*.

Notice that the results are a little more refined as well as consistent.

## Gathering five events from four classes

C File            **five\_events.c**

Command        **tracelogger -d1 -n 3 -f  
                 five\_events.results**

Running C File

**five\_events**

Sample result file

**five\_events.results.txt**

Viewing        **traceprinter -f five\_events.results**

Now that we can gather specific classes of events, we'll refine our search even further and gather only five specific types of events from four classes. We've now begun to selectively pick and choose events — the massive amount of data is now much more manageable.

## Gathering kernel calls

C File        **ker\_calls.c**

Command      **tracelogger -d1 -n 3 -f  
ker\_calls.all.results**

Running C File

**ker\_calls**

Sample result file

**ker\_calls.all.results.txt**

Viewing      **traceprinter -f ker\_calls.all.results**

The kernel calls are arguably the most important class of calls. This example shows not only filtering, but also the arguments intercepted by the Instrumented Kernel. In its base form, this program is similar to the **one\_class.c** example that gathered only one class.

C File        **ker\_calls.c**

Command      **tracelogger -d1 -n 3 -f  
ker\_calls.14.results**

Running C File

**ker\_calls -n 14**

Sample result file

**ker\_calls.14.results.txt**

Viewing      **traceprinter -f ker\_calls.14.results**

Our **C File** has an extra feature in it that lets us view only one type of kernel call — the number 14 signifies `_KER_MSG_RECEIVE`. For a full list of the values associated with the `-n` option, see `/usr/include/sys/kercalls.h`.

## Event handling - simple

C File            `eh_simple.c`

Command         `tracelogger -d1 -n 3 -f`  
                  `eh_simple.results`

Running C File

`eh_simple`

Sample result file

`eh_simple.results.txt`

Viewing         `traceprinter -f eh_simple.results`

Now, two events from two different classes are intercepted. Each event has an event handler attached to it; the event handlers are closing and opening the stream.

This is an important example because it demonstrates the use of the dynamic rules filter to perform tasks beyond basic filtering.

## User event - simple

C File            `usr_event_simple.c`

Command         `tracelogger -d1 -n 3 -f`  
                  `usr_event_simple.results`

Running C File

`usr_event_simple`

Sample result file

`usr_event_simple.results.txt`

```
Viewing      traceprinter -f
             usr_event_simple.results
```

This example demonstrates the insertion of a user event into the event stream.

Being able to insert “artificial” events allows you to “flag” events or “bracket” groups of events to isolate them for study. It’s also useful for inserting internal, customized information into the event stream.

### Header file: `instrex.h`

```
/*
 * Copyright 2003, QNX Software Systems Ltd. All Rights Reserved.
 *
 * This source code may contain confidential information of QNX Software
 * Systems Ltd. (QSSL) and its licensors. Any use, reproduction,
 * modification, disclosure, distribution or transfer of this software,
 * or any software which includes or is based upon any of this code, is
 * prohibited unless expressly authorized by QSSL by written agreement. For
 * more information (including whether this source code file has been
 * published) please email licensing@qnx.com.
 */

/*
 * instrex.h instrumentation examples - public definitions
 *
 */

#ifndef __INSTREX_H_INCLUDED

#include <errno.h>
#include <stdio.h>
#include <string.h>

/*
 * Supporting macro that intercepts and prints a possible
 * error states during calling TraceEvent(...)
 *
 * Call TRACE_EVENT(TraceEvent(...)) <=> TraceEvent(...)
 *
 */
#define TRACE_EVENT(prog_name, trace_event) \
if((int)((trace_event))=(-1)) \
{ \
(void) fprintf \
( \
stderr, \
"%s: line:%d function call TraceEvent() failed, errno(%d): %s\n", \
prog_name, \
__LINE__, \
errno, \
strerror(errno) \
); \
}
```

```

\
return (-1); \
}

/*
 * Prints error message
 */
#define TRACE_ERROR_MSG(prog_name, msg) \
(void) fprintf(stderr,"%s: %s\n", prog_name, msg)

#define __INSTREX_H_INCLUDED
#endif

```

### C File: **all\_classes.c**

```

/*
 * Copyright 2003, QNX Software Systems Ltd. All Rights Reserved.
 *
 * This source code may contain confidential information of QNX Software
 * Systems Ltd. (QSSL) and its licensors. Any use, reproduction,
 * modification, disclosure, distribution or transfer of this software,
 * or any software which includes or is based upon any of this code, is
 * prohibited unless expressly authorized by QSSL by written agreement. For
 * more information (including whether this source code file has been
 * published) please email licensing@qnx.com.
 */

#ifdef __USAGE
%C - instrumentation example

%C - example that illustrates the very basic use of
the TraceEvent() kernel call and the instrumentation
module with tracelogger in a daemon mode.

All classes and their events are included and monitored.

In order to use this example, start the tracelogger
in the daemon mode as:

tracelogger -n iter_number -dl

with iter_number = your choice of 1 through 10

After executing the example, the tracelogger (daemon)
will log the specified number of iterations and then
terminate. There are no messages printed upon successful
completion of the example. You can view the intercepted
events with the traceprinter utility.

See accompanied documentation and comments within
the example source code for more explanations.
#endif

#include <sys/trace.h>

#include "instrex.h"

```

```

int main(int argc, char **argv)
{
/*
 * Just in case, turn off all filters, since we
 * don't know their present state - go to the
 * known state of the filters.
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_DELALLCLASSES));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_KERCALL));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_KERCALL));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_THREAD));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_THREAD));

/*
 * Set fast emitting mode for all classes and
 * their events.
 * Wide emitting mode could have been
 * set instead, using:
 *
 * TraceEvent(_NTO_TRACE_SETALLCLASSESWIDE)
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_SETALLCLASSESFAST));

/*
 * Intercept all event classes
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_ADDALLCLASSES));

/*
 * Start tracing process
 *
 * During the tracing process, the tracelogger (which
 * is being executed in a daemon mode) will log all events.
 * The number of full logged iterations is user specified.
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_START));

/*
 * The main() of this execution flow returns.
 * However, the main() function of the tracelogger
 * will return after registering the specified number
 * of events.
 */
return (0);
}

```

### Sample result file: all\_classes.results.txt

```

TRACEPRINTER version 0.97
TRACEPARSER LIBRARY version 0.98
-- HEADER FILE INFORMATION --
TRACE_FILE_NAME:: /dev/shmem/tracebuffer
TRACE_DATE:: Fri Aug 17 09:08:06 2001
TRACE_VER_MAJOR:: 0
TRACE_VER_MINOR:: 97

```

```

TRACE_LITTLE_ENDIAN:: TRUE
TRACE_ENCODING:: 16 byte events
TRACE_BOOT_DATE:: Fri Aug 17 09:02:16 2001
TRACE_CYCLES_PER_SEC:: 132980400
TRACE_CPU_NUM:: 1
TRACE_SYSNAME:: QNX
TRACE_NODENAME:: localhost
TRACE_SYS_RELEASE:: 6.1.0
TRACE_SYS_VERSION:: 2001/08/15-08:15:15
TRACE_MACHINE:: x86pc
TRACE_SYSPAGE_LEN:: 2248
-- KERNEL EVENTS --
t:0xbcb12bdf CPU:00 CONTROL :TIME msb:0x0000000b, lsb(offset):0xbcb128b3
t:0xbcb135fb CPU:00 PROCESS :PROCCREATE_NAME
      ppid:0
      pid:1
      name:/home/mmacies/instrumentation/x86/procnto-instr
.
.
.
t:0xbcb1db36 CPU:00 THREAD :THRUNNING      pid:98319 tid:1
t:0xbcb1e090 CPU:00 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000 empty:0x00000000
t:0xbcb1f03c CPU:00 KER_CALL:THREAD_DESTROY/47 tid:-1 status_p:0
t:0xbcb1f870 CPU:00 KER_EXIT:THREAD_DESTROY/47 ret_val:0x00000030 empty:0x00000000
t:0xbcb1fab2 CPU:00 KER_CALL:THREAD_DESTROYALL/48 empty:0x00000000 empty:0x00000000
t:0xbcb1fadb CPU:00 THREAD :THDESTROY      pid:98319 tid:1
t:0xbcb20055 CPU:00 THREAD :THDEAD          pid:98319 tid:1
t:0xbcb2021e CPU:00 THREAD :THRUNNING      pid:7 tid:5
t:0xbcb21262 CPU:00 THREAD :THREADY        pid:1 tid:5
t:0xbcb21d61 CPU:00 KER_EXIT:MSG_RECEIVEV/14 rcvid:0x00000000 rmsg:"" (0x00000000)
t:0xbcb234e8 CPU:00 INT_ENTR:0x00000000 (0) inkernel=0x00000001
t:0xbcb238b4 CPU:00 INT_EXIT:0x00000000 (0) inkernel=0x00000001
t:0xbcb2905f CPU:00 KER_CALL:INTERRUPT_UNMASK/60 intr:10 id:5
t:0xbcb297e2 CPU:00 KER_EXIT:INTERRUPT_UNMASK/60 mask_level:0x00000000 empty:0x00000000
t:0xbcb29fca CPU:00 THREAD :THRECEIVE      pid:7 tid:5
t:0xbcb2a177 CPU:00 THREAD :THRUNNING      pid:7 tid:11
t:0xbcb2a4c8 CPU:00 KER_EXIT:SYNC_CONDVAR_WAIT/82 ret_val:0 empty:0x00000000
t:0xbcb2b8de CPU:00 KER_CALL:SYNC_CONDVAR_SIGNAL/83 sync_p:807144c all:0
t:0xbcb2bf6c CPU:00 THREAD :THREADY        pid:7 tid:8
t:0xbcb2c256 CPU:00 KER_EXIT:SYNC_CONDVAR_SIG/83 ret_val:0 empty:0x00000000
t:0xbcb2c8ec CPU:00 KER_CALL:SYNC_CONDVAR_WAIT/82 sync_p:8071b90 mutex_p:8071b88
t:0xbcb2cdd4 CPU:00 THREAD :THCONDVAR      pid:7 tid:11
t:0xbcb2cf82 CPU:00 THREAD :THRUNNING      pid:1 tid:5
t:0xbcb2d438 CPU:00 INT_CALL:KER_MSG_RECEIVEV/14
t:0xbcb2e00c CPU:00 KER_CALL:RING0/02 func_p:ff836f16 arg_p:e3fd2ee0
t:0xbcb2e470 CPU:00 KER_EXIT:RING0/02 ret_val:0xe313c7b8 empty:0x00000000
t:0xbcb2e93c CPU:00 KER_CALL:RING0/02 func_p:ff83562c arg_p:e313c8ac
t:0xbcb2eb46 CPU:00 KER_EXIT:RING0/02 ret_val:0x00000002 empty:0x00000000
t:0xbcb2eee8 CPU:00 KER_CALL:RING0/02 func_p:ff836f16 arg_p:e3fd2e54
.
.
.
t:0xbcd2a81c CPU:00 INT_ENTR:0x00000000 (0) inkernel=0x00000081
t:0xbcd2a9a4 CPU:00 INT_EXIT:0x00000000 (0) inkernel=0x00000081
t:0xbcd2ac29 CPU:00 KER_EXIT:RING0/02 ret_val:0x00000000 empty:0x00000000
t:0xbcd2ae9a CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0

```

**C File: one\_class.c**

```
/*
 * Copyright 2003, QNX Software Systems Ltd. All Rights Reserved.
 *
 * This source code may contain confidential information of QNX Software
 * Systems Ltd. (QSSL) and its licensors. Any use, reproduction,
 * modification, disclosure, distribution or transfer of this software,
 * or any software which includes or is based upon any of this code, is
 * prohibited unless expressly authorized by QSSL by written agreement. For
 * more information (including whether this source code file has been
 * published) please email licensing@qnx.com.
 */

#ifdef __USAGE
%C - instrumentation example

%C - example that illustrates the very basic use of
the TraceEvent() kernel call and the instrumentation
module with tracelogger in a daemon mode.

Only events from the thread class (_NTO_TRACE_THREAD)
are monitored (intercepted).

In order to use this example, start the tracelogger
in the daemon mode as:

tracelogger -n iter_number -dl

with iter_number = your choice of 1 through 10

After executing the example, the tracelogger (daemon)
will log the specified number of iterations and then
terminate. There are no messages printed upon successful
completion of the example. You can view the intercepted
events with the traceprinter utility.

See accompanied documentation and comments within
the example source code for more explanations.
#endif

#include <sys/trace.h>

#include "instrex.h"

int main(int argc, char **argv)
{
/*
 * Just in case, turn off all filters, since we
 * don't know their present state - go to the
 * known state of the filters.
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_DELALLCLASSES));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_KERCALL));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_KERCALL));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_THREAD));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_THREAD));

/*
```

```

* Intercept only thread events
*/
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_ADDCLASS, _NTO_TRACE_THREAD));

/*
* Start tracing process
*
* During the tracing process, the tracelogger (which
* is being executed in daemon mode) will log all events.
* The number of full logged iterations is user specified.
*/
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_START));

/*
* The main() of this execution flow returns.
* However, the main() function of the tracelogger
* will return after registering the specified number
* of events.
*/
return (0);
}

```

### Sample result file: **one\_class.results.txt**

```

TRACEPRINTER version 0.97
TRACEPARSER LIBRARY version 0.98
-- HEADER FILE INFORMATION --
TRACE_FILE_NAME:: /dev/shmem/tracebuffer
TRACE_DATE:: Fri Aug 17 09:15:34 2001
TRACE_VER_MAJOR:: 0
TRACE_VER_MINOR:: 97
TRACE_LITTLE_ENDIAN:: TRUE
TRACE_ENCODING:: 16 byte events
TRACE_BOOT_DATE:: Fri Aug 17 09:02:16 2001
TRACE_CYCLES_PER_SEC:: 132980400
TRACE_CPU_NUM:: 1
TRACE_SYSNAME:: QNX
TRACE_NODENAME:: localhost
TRACE_SYS_RELEASE:: 6.1.0
TRACE_SYS_VERSION:: 2001/08/15-08:15:15
TRACE_MACHINE:: x86pc
TRACE_SYSPAGE_LEN:: 2248
-- KERNEL EVENTS --
t:0x9bebdaf3 CPU:00 THREAD :THCREATE pid:1 tid:1
t:0x9bebdd74 CPU:00 THREAD :THREADY pid:1 tid:1
t:0x9bebde66 CPU:00 THREAD :THCREATE pid:1 tid:2
t:0x9bebdf46 CPU:00 THREAD :THRECEIVE pid:1 tid:2
t:0x9bebe05c CPU:00 THREAD :THCREATE pid:1 tid:3
.
.
.
t:0x9bebe120 CPU:00 THREAD :THRECEIVE pid:1 tid:3
t:0x9bebe25e CPU:00 THREAD :THCREATE pid:1 tid:4
t:0x9bebe31c CPU:00 THREAD :THRECEIVE pid:1 tid:4
t:0x9bebe42c CPU:00 THREAD :THCREATE pid:1 tid:5
t:0x9bebe4ea CPU:00 THREAD :THRECEIVE pid:1 tid:5

```

```

t:0x9bebe5e4 CPU:00 THREAD :THCREATE      pid:1 tid:6
t:0x9bebe6a2 CPU:00 THREAD :THRECEIVE    pid:1 tid:6
t:0x9c436edf CPU:00 THREAD :THREADY      pid:8 tid:5
.
.
.
t:0x9c437a15 CPU:00 THREAD :THRECEIVE    pid:7 tid:2
t:0x9c437be0 CPU:00 THREAD :THRUNNING    pid:8 tid:5

```

## C File: five\_events.c

```

/*
 * Copyright 2003, QNX Software Systems Ltd. All Rights Reserved.
 *
 * This source code may contain confidential information of QNX Software
 * Systems Ltd. (QSSL) and its licensors. Any use, reproduction,
 * modification, disclosure, distribution or transfer of this software,
 * or any software which includes or is based upon any of this code, is
 * prohibited unless expressly authorized by QSSL by written agreement. For
 * more information (including whether this source code file has been
 * published) please email licensing@qnx.com.
 */

#ifdef __USAGE
%C - instrumentation example

%C - example that illustrates the very basic use of
the TraceEvent() kernel call and the instrumentation
module with tracelogger in a daemon mode.

Only five events from four classes are included and
monitored. Class _NTO_TRACE_KERCALL is intercepted
in a wide emitting mode.

In order to use this example, start the tracelogger
in the daemon mode as:

tracelogger -n iter_number -dl

with iter_number = your choice of 1 through 10

After executing the example, the tracelogger (daemon)
will log the specified number of iterations and then
terminate. There are no messages printed upon successful
completion of the example. You can view the intercepted
events with the traceprinter utility.

See accompanied documentation and comments within
the example source code for more explanations.
#endif

#include <sys/trace.h>
#include <sys/kercalls.h>

#include "instrex.h"

int main(int argc, char **argv)

```

```

{
/*
 * Just in case, turn off all filters, since we
 * don't know their present state - go to the
 * known state of the filters.
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_DELALLCLASSES));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_KERCALL));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_KERCALL));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_THREAD));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_THREAD));

/*
 * Set wide emitting mode
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_SETALLCLASSESWIDE));

/*
 * Intercept two events from class _NTO_TRACE_THREAD
 */
TRACE_EVENT
(
  argv[0],
  TraceEvent(_NTO_TRACE_ADDEVENT, _NTO_TRACE_THREAD, _NTO_TRACE_THRUNNING)
);
TRACE_EVENT
(
  argv[0],
  TraceEvent(_NTO_TRACE_ADDEVENT, _NTO_TRACE_THREAD, _NTO_TRACE_THCREATE)
);

/*
 * Intercept one event from class _NTO_TRACE_PROCESS
 */
TRACE_EVENT
(
  argv[0],
  TraceEvent(_NTO_TRACE_ADDEVENT, _NTO_TRACE_PROCESS, _NTO_TRACE_PROCCREATE_NAME)
);

/*
 * Intercept one event from class _NTO_TRACE_INTENTER
 */
TRACE_EVENT
(
  argv[0],
  TraceEvent(_NTO_TRACE_ADDEVENT, _NTO_TRACE_INTENTER, _NTO_TRACE_INTFIRST)
);

/*
 * Intercept one event from class _NTO_TRACE_KERCALEXIT,
 * event __KER_MSG_READV.
 */
TRACE_EVENT
(
  argv[0],
  TraceEvent(_NTO_TRACE_ADDEVENT, _NTO_TRACE_KERCALEXIT, __KER_MSG_READV)
);
/*

```

```

* Start tracing process
*
* During the tracing process, the tracelogger (which
* is being executed in a daemon mode) will log all events.
* The number of full logged iterations is user specified.
*/
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_START));

/*
* The main() of this execution flow returns.
* However, the main() function of the tracelogger
* will return after registering the specified number
* of events.
*/
return (0);
}

```

### Sample result file: five\_events.results.txt

```

TRACEPRINTER version 0.97
TRACEPARSER LIBRARY version 0.98
-- HEADER FILE INFORMATION --
TRACE_FILE_NAME:: /dev/shmem/tracebuffer
TRACE_DATE:: Fri Aug 17 09:15:51 2001
TRACE_VER_MAJOR:: 0
TRACE_VER_MINOR:: 97
TRACE_LITTLE_ENDIAN:: TRUE
TRACE_ENCODING:: 16 byte events
TRACE_BOOT_DATE:: Fri Aug 17 09:02:16 2001
TRACE_CYCLES_PER_SEC:: 132980400
TRACE_CPU_NUM:: 1
TRACE_SYSNAME:: QNX
TRACE_NODENAME:: localhost
TRACE_SYS_RELEASE:: 6.1.0
TRACE_SYS_VERSION:: 2001/08/15-08:15:15
TRACE_MACHINE:: x86pc
TRACE_SYSPAGE_LEN:: 2248
-- KERNEL EVENTS --
t:0x3abf47cf CPU:00 PROCESS :PROCCREATE_NAME
      ppid:0
      pid:1
      name:/home/mmacies/instrumentation/x86/procnto-instr
t:0x3abf50b1 CPU:00 THREAD :THCREATE      pid:1 tid:1
t:0x3abf5208 CPU:00 THREAD :THCREATE      pid:1 tid:2
t:0x3abf5326 CPU:00 THREAD :THCREATE      pid:1 tid:3
t:0x3abf546a CPU:00 THREAD :THCREATE      pid:1 tid:4
t:0x3abf5554 CPU:00 THREAD :THCREATE      pid:1 tid:5
t:0x3abf5654 CPU:00 THREAD :THCREATE      pid:1 tid:6
t:0x3abf57d8 CPU:00 THREAD :THCREATE      pid:1 tid:7
t:0x3abf58d8 CPU:00 THREAD :THCREATE      pid:1 tid:8
.
.
.
t:0x3abfd7da CPU:00 THREAD :THRUNNING     pid:184335 tid:1
t:0x3abff400 CPU:00 THREAD :THRUNNING     pid:7 tid:11
t:0x3ac02c55 CPU:00 THREAD :THRUNNING     pid:1 tid:5

```

```

t:0x3ac051d8 CPU:00 THREAD :THCREATE      pid:184335 tid:1
t:0x3ac0535b CPU:00 THREAD :THRUNNING     pid:184335 tid:1
t:0x3ac0a687 CPU:00 THREAD :THRUNNING     pid:4 tid:1
t:0x3ac0e597 CPU:00 KER_EXIT:MSG_READV/16
                    rbytes:47
                    rmsg:"" (0x6f72500a 0x73736563 0x34383120)
t:0x3ac1ad22 CPU:00 INT_ENTR:0x00000000 (0) inkernel=0x00000001
t:0x3ac3b3fc CPU:00 INT_ENTR:0x00000000 (0) inkernel=0x00000001
t:0x3ac5baba CPU:00 INT_ENTR:0x00000000 (0) inkernel=0x00000001
t:0x3ac5bf32 CPU:00 THREAD :THRUNNING     pid:1 tid:5
t:0x3ac5c9f8 CPU:00 THREAD :THRUNNING     pid:4 tid:1
.
.
.
t:0x3b8a6de3 CPU:00 THREAD :THRUNNING     pid:7 tid:11
t:0x3b8a7d4e CPU:00 INT_ENTR:0x00000000 (0) inkernel=0x00000001
t:0x3b8a984c CPU:00 THREAD :THRUNNING     pid:7 tid:8

```

## C File: `ker_calls.c`

```

/*
 * Copyright 2003, QNX Software Systems Ltd. All Rights Reserved.
 *
 * This source code may contain confidential information of QNX Software
 * Systems Ltd. (QSSL) and its licensors. Any use, reproduction,
 * modification, disclosure, distribution or transfer of this software,
 * or any software which includes or is based upon any of this code, is
 * prohibited unless expressly authorized by QSSL by written agreement. For
 * more information (including whether this source code file has been
 * published) please email licensing@qnx.com.
 */

#ifdef __USAGE
%C - instrumentation example

%C - [-n num]

%C - example that illustrates the very basic use of
the TraceEvent() kernel call and the instrumentation
module with tracelogger in a daemon mode.

All thread states and all/one (specified) kernel
call number are intercepted. The kernel call(s)
is(are) intercepted in wide emitting mode.

Options:
-n <num> kernel call number to be intercepted
  (default is all)

In order to use this example, start the tracelogger
in the daemon mode as:

tracelogger -n iter_number -dl

with iter_number = your choice of 1 through 10

After executing the example, the tracelogger (daemon)

```

will log the specified number of iterations and then terminate. There are no messages printed upon successful completion of the example. You can view the intercepted events with the `traceprinter` utility.

See accompanied documentation and comments within the example source code for more explanations.

```
#endif

#include <sys/trace.h>
#include <unistd.h>
#include <stdlib.h>

#include "instrex.h"

int main(int argc, char **argv)
{
    int arg_var;          /* input arguments parsing support */
    int call_num=(-1); /* kernel call number to be intercepted */

    /* Parse command line arguments
     *
     * - get optional kernel call number
     */
    while((arg_var=getopt(argc, argv,"n:"))!=(-1)) {
        switch(arg_var)
        {
            case 'n': /* get kernel call number */
                call_num = strtoul(optarg, NULL, 10);
                break;
            default: /* unknown */
                TRACE_ERROR_MSG
                (
                    argv[0],
                    "error parsing command-line arguments - exiting\n"
                );
        }

        return (-1);
    }
}

/*
 * Just in case, turn off all filters, since we
 * don't know their present state - go to the
 * known state of the filters.
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_DELALLCLASSES));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_KERCALL));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_KERCALL));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_THREAD));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_THREAD));

/*
 * Set wide emitting mode for all classes and
 * their events.
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_SETALLCLASSESWIDE));

/*
 * Intercept _NTO_TRACE_THREAD class
 */
```

```

    * We need it to know the state of the active thread.
    */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_ADDCLASS, _NTO_TRACE_THREAD));

/*
 * Add all/one kernel call
 */
if(call_num != (-1)) {
TRACE_EVENT
(
    argv[0],
    TraceEvent(_NTO_TRACE_ADDEVENT, _NTO_TRACE_KERNEL, call_num)
);
} else {
TRACE_EVENT
(
    argv[0],
    TraceEvent(_NTO_TRACE_ADDCLASS, _NTO_TRACE_KERNEL)
);
}

/*
 * Start tracing process
 *
 * During the tracing process, the tracelogger (which
 * is being executed in a daemon mode) will log all events.
 * The number of full logged iterations is user specified.
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_START));

/*
 * The main() of this execution flow returns.
 * However, the main() function of the tracelogger
 * will return after registering the specified number
 * of events.
 */
return (0);
}

```

### Sample result file: **ker\_calls.all.results.txt**

```

TRACEPRINTER version 0.97
TRACEPARSER LIBRARY version 0.98
-- HEADER FILE INFORMATION --
    TRACE_FILE_NAME:: /dev/shmem/tracebuffer
    TRACE_DATE:: Fri Aug 17 09:17:00 2001
    TRACE_VER_MAJOR:: 0
    TRACE_VER_MINOR:: 97
    TRACE_LITTLE_ENDIAN:: TRUE
    TRACE_ENCODING:: 16 byte events
    TRACE_BOOT_DATE:: Fri Aug 17 09:02:16 2001
TRACE_CYCLES_PER_SEC:: 132980400
    TRACE_CPU_NUM:: 1
    TRACE_SYSNAME:: QNX
    TRACE_NODENAME:: localhost
    TRACE_SYS_RELEASE:: 6.1.0

```

```

TRACE_SYS_VERSION:: 2001/08/15-08:15:15
TRACE_MACHINE:: x86pc
TRACE_SYSPAGE_LEN:: 2248
-- KERNEL EVENTS --
t:0x35b45a23 CPU:00 THREAD :THCREATE      pid:1 tid:1
t:0x35b45b6e CPU:00 THREAD :THREADY      pid:1 tid:1
t:0x35b45c86 CPU:00 THREAD :THCREATE      pid:1 tid:2
.
.
t:0x35b45d54 CPU:00 THREAD :THRECEIVE    pid:1 tid:2
t:0x35b45e8c CPU:00 THREAD :THCREATE      pid:1 tid:3
t:0x35b45f50 CPU:00 THREAD :THRECEIVE    pid:1 tid:3
t:0x35b4608e CPU:00 THREAD :THCREATE      pid:1 tid:4
t:0x35b4614c CPU:00 THREAD :THRECEIVE    pid:1 tid:4
t:0x35b4622e CPU:00 THREAD :THCREATE      pid:1 tid:5
t:0x35b462ec CPU:00 THREAD :THRECEIVE    pid:1 tid:5
t:0x35b463e6 CPU:00 THREAD :THCREATE      pid:1 tid:6
t:0x35b464a4 CPU:00 THREAD :THRECEIVE    pid:1 tid:6
t:0x35b4663a CPU:00 THREAD :THCREATE      pid:1 tid:7
t:0x35b466f8 CPU:00 THREAD :THRECEIVE    pid:1 tid:7
t:0x35b467f2 CPU:00 THREAD :THCREATE      pid:1 tid:8
t:0x35b468b0 CPU:00 THREAD :THRECEIVE    pid:1 tid:8
t:0x35b46b20 CPU:00 THREAD :THCREATE      pid:2 tid:1
.
.
t:0x35bbdee9 CPU:00 THREAD :THREADY      pid:4 tid:1
t:0x35bbe032 CPU:00 THREAD :THRUNNING    pid:1 tid:5
t:0x35bbe97a CPU:00 KER_CALL:MSG_RECEIVEV/14 chid:0x00000001 rparts:2080
t:0x35bbec68 CPU:00 THREAD :THRECEIVE    pid:1 tid:5
t:0x35bbe10 CPU:00 THREAD :THRUNNING    pid:4 tid:1
t:0x35c2f825 CPU:00 THREAD :THREADY      pid:208911 tid:1
t:0x35c2ff68 CPU:00 KER_CALL:MSG_RECEIVEV/14 chid:0x00000001 rparts:2077
t:0x35c30190 CPU:00 THREAD :THRECEIVE    pid:4 tid:1
t:0x35c302c6 CPU:00 THREAD :THRUNNING    pid:208911 tid:1
t:0x35c32979 CPU:00 THREAD :THREPLY     pid:208911 tid:1
t:0x35c32c6f CPU:00 THREAD :THRUNNING    pid:4 tid:1
t:0x35c33487 CPU:00 KER_EXIT:MSG_RECEIVEV/14
                rcvid:0x00000012
                rmsg:"" (0x00040116 0x0000002d 0x00000000)
                info->nd:0
                info->srcnd:0
                info->pid:208911
                info->tid:1
                info->chid:1
                info->scoid:1073741842
                info->coid:0
                info->msglen:4
                info->srcmsglen:4
                info->dstmsglen:2147483647
                info->priority:15
                info->flags:0
                info->reserved:0
t:0x35c35919 CPU:00 THREAD :THREADY      pid:208911 tid:1
t:0x35c36032 CPU:00 KER_CALL:MSG_RECEIVEV/14 chid:0x00000001 rparts:2077
.
.
t:0x35dalb25 CPU:00 THREAD :THRUNNING    pid:7 tid:5

```

```
t:0x35da1c4e CPU:00 THREAD :THREADY      pid:1 tid:1
```

### Sample result file: ker\_calls.14.results.txt

```
TRACEPRINTER version 0.97
TRACEPARSER LIBRARY version 0.98
-- HEADER FILE INFORMATION --
TRACE_FILE_NAME:: /dev/shmem/tracebuffer
TRACE_DATE:: Wed Aug 22 09:52:49 2001
TRACE_VER_MAJOR:: 0
TRACE_VER_MINOR:: 97
TRACE_LITTLE_ENDIAN:: TRUE
TRACE_ENCODING:: 16 byte events
TRACE_BOOT_DATE:: Tue Aug 21 14:03:25 2001
TRACE_CYCLES_PER_SEC:: 132961600
TRACE_CPU_NUM:: 1
TRACE_SYSNAME:: QNX
TRACE_NODENAME:: localhost
TRACE_SYS_RELEASE:: 6.1.0
TRACE_SYS_VERSION:: 2001/08/21-14:05:41
TRACE_MACHINE:: x86pc
TRACE_SYSPAGE_LEN:: 2248
-- KERNEL EVENTS --
t:0x4c091677 CPU:00 THREAD :THCREATE      pid:1 tid:1
t:0x4c0917ec CPU:00 THREAD :THREADY      pid:1 tid:1
t:0x4c098b66 CPU:00 THREAD :THDEAD      pid:12302 tid:1
.
.
.
t:0x4c099ed2 CPU:00 THREAD :THREADY      pid:1 tid:1
t:0x4c09a422 CPU:00 INT_CALL:KER_MSG_RECEIVEV/14
t:0x4c09c872 CPU:00 THREAD :THCREATE      pid:12302 tid:1
t:0x4c09c97c CPU:00 THREAD :THWAITTHREAD pid:1 tid:2
t:0x4c09cabf CPU:00 THREAD :THRUNNING    pid:12302 tid:1
t:0x4c09d17c CPU:00 THREAD :THREADY      pid:1 tid:2
t:0x4c0a13db CPU:00 THREAD :THREPLY     pid:12302 tid:1
t:0x4c0a1618 CPU:00 THREAD :THREADY      pid:4 tid:1
t:0x4c0a1786 CPU:00 THREAD :THRUNNING    pid:4 tid:1
t:0x4c0a205f CPU:00 KER_EXIT:MSG_RECEIVEV/14
                    rcvid:0x00000011
                    rmsg:"" (0x00100102 0x0000002c 0x00000000)
                    info->nd:0
                    info->srcnd:0
                    info->pid:12302
                    info->tid:1
                    info->chid:1
                    info->scoid:1073741841
                    info->coid:2
                    info->msglen:60
                    info->srcmsglen:60
                    info->dstmsglen:2147483647
                    info->priority:15
                    info->flags:0
                    info->reserved:0
t:0x4c0b89e9 CPU:00 THREAD :THREADY      pid:4 tid:1
t:0x4c0b8b2e CPU:00 THREAD :THRUNNING    pid:1 tid:2
.
.
```

```

.
.
t:0x5a0dc58d CPU:00 THREAD :THRUNNING      pid:7 tid:6
t:0x5a0dc6d0 CPU:00 THREAD :THREADY       pid:1 tid:1
t:0x5a0dcb88 CPU:00 KER_EXIT:MSG_RECEIVEV/14
      rcvid:0x00000000
      rmsg:"" (0x00000000 0x00000000 0x00000030)
      info->nd:0
      info->srcnd:0
      info->pid:0
      info->tid:0
      info->chid:0
      info->scoid:0
      info->coid:0
      info->msglen:0
      info->srcmsglen:0
      info->dstmsglen:0
      info->priority:0
      info->flags:0
      info->reserved:0
t:0x5a0dd3f8 CPU:00 KER_CALL:MSG_RECEIVEV/14 chid:0x00000009 rparts:16
t:0x5a0dd58a CPU:00 THREAD :THRECEIVE     pid:7 tid:6
t:0x5a0dd6f6 CPU:00 THREAD :THRUNNING     pid:1 tid:1

```

## C File: eh\_simple.c

```

/*
 * Copyright 2003, QNX Software Systems Ltd. All Rights Reserved.
 *
 * This source code may contain confidential information of QNX Software
 * Systems Ltd. (QSSL) and its licensors. Any use, reproduction,
 * modification, disclosure, distribution or transfer of this software,
 * or any software which includes or is based upon any of this code, is
 * prohibited unless expressly authorized by QSSL by written agreement. For
 * more information (including whether this source code file has been
 * published) please email licensing@qnx.com.
 */

#ifdef __USAGE
%C - instrumentation example

%C - example that illustrates the very basic use of
the TraceEvent() kernel call and the instrumentation
module with tracelogger in a daemon mode.

Two events from two classes are included and monitored
interchangeably. The flow control of monitoring the
specified events is controlled with attached event
handlers.

In order to use this example, start the tracelogger
in the daemon mode as:

tracelogger -n 1 -d1

After executing the example, the tracelogger (daemon)
will log the specified number of iterations and then

```

```
terminate. There are no messages printed upon successful
completion of the example. You can view the intercepted
events with the traceprinter utility.

See accompanied documentation and comments within
the example source code for more explanations.
#endif

#include <unistd.h>
#include <sys/trace.h>
#include <sys/kercalls.h>

#include "instrex.h"

/*
 * Prepare event structure where the event data will be
 * stored and passed to an event handler.
 */
event_data_t e_d_1;
_uint32t    data_array_1[20]; /* 20 elements for potential args. */

event_data_t e_d_2;
_uint32t    data_array_2[20]; /* 20 elements for potential args. */

/*
 * Global state variable that controls the
 * event flow between two events
 */
int g_state;

/*
 * Event handler attached to the event "ring0"
 * from the _NTO_TRACE_KERCALL class.
 */
int call_ring0_eh(event_data_t* e_d)
{
    if(g_state) {
        g_state = !g_state;
        return (1);
    }

    return (0);
}

/*
 * Event handler attached to the event _NTO_TRACE_THRUNNING
 * from the _NTO_TRACE_THREAD (thread) class.
 */
int thread_run_eh(event_data_t* e_d)
{
    if(!g_state) {
        g_state = !g_state;
        return (1);
    }

    return (0);
}

int main(int argc, char **argv)
```

```

{
/*
 * First fill arrays inside event data structures
 */
e_d_1.data_array = data_array_1;
e_d_2.data_array = data_array_2;

/*
 * Just in case, turn off all filters, since we
 * don't know their present state - go to the
 * known state of the filters.
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_DELALLCLASSES));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_KERCALL));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASPID, _NTO_TRACE_KERCALL));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_THREAD));
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASPID, _NTO_TRACE_THREAD));

/*
 * Set fast emitting mode
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_SETALLCLASSESFAST));

/*
 * Obtain I/O privity before adding event handlers
 */
if (ThreadCtl(_NTO_TCTL_IO, 0) != EOK) { /* obtain I/O privity */
(void) fprintf(stderr, "argv[0]: Fail to obtain I/O privity - root privileges\n");

return (-1);
}

/*
 * Intercept one event from class _NTO_TRACE_KERCALL,
 * event __KER_MSG_READV.
 */
TRACE_EVENT
(
argv[0],
TraceEvent(_NTO_TRACE_ADDEVENT, _NTO_TRACE_KERCALLENTER, __KER_RING0)
);

/*
 * Add event handler to the event "ring0"
 * from _NTO_TRACE_KERCALL class.
 */
TRACE_EVENT
(
argv[0],
TraceEvent(_NTO_TRACE_ADDEVENTHANDLER,
_NTO_TRACE_KERCALLENTER, __KER_RING0, call_ring0_ah, &e_d_1)
);

/*
 * Intercept one event from class _NTO_TRACE_THREAD
 */
TRACE_EVENT
(
argv[0],
TraceEvent(_NTO_TRACE_ADDEVENT, _NTO_TRACE_THREAD, _NTO_TRACE_THRUNNING)
)

```

```

);

/*
 * Add event handler to the _NTO_TRACE_THRUNNING event
 * from the _NTO_TRACE_THREAD (thread) class.
 */
TRACE_EVENT
(
  argv[0],
  TraceEvent(_NTO_TRACE_ADDEVENTHANDLER,
             _NTO_TRACE_THREAD, _NTO_TRACE_THRUNNING, thread_run_ah, &e_d_2)
);

/*
 * Start tracing process
 *
 * During the tracing process, the tracelogger (which
 * is being executed in a daemon mode) will log all events.
 * The number of full logged iterations has been specified
 * to be 1.
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_START));

/*
 * During one second collect all events
 */
(void) sleep(1);

/*
 * Stop tracing process by closing the event stream.
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_STOP));

/*
 * Flush the internal buffer since the number
 * of stored events could be less than
 * "high water mark" of one buffer (715 events).
 *
 * The tracelogger will probably terminate at
 * this point, since it has been executed with
 * one iteration (-n 1 "option").
 */
TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_FLUSHBUFFER));

/*
 * Delete event handlers before exiting to avoid execution
 * in the missing address space.
 */
TRACE_EVENT
(
  argv[0],
  TraceEvent(_NTO_TRACE_DELEVENTHANDLER, _NTO_TRACE_KERNELCENTER, __KER_RING0)
);
TRACE_EVENT
(
  argv[0],
  TraceEvent(_NTO_TRACE_DELEVENTHANDLER, _NTO_TRACE_THREAD, _NTO_TRACE_THRUNNING)
);

```

```

/*
 * Wait one second before terminating to hold the address space
 * of the event handlers.
 */
(void) sleep(1);

return (0);
}

```

### Sample result file: eh\_sample.results.txt

```

TRACEPRINTER version 0.97
TRACEPARSER LIBRARY version 0.98
-- HEADER FILE INFORMATION --
  TRACE_FILE_NAME:: /dev/shmem/tracebuffer
  TRACE_DATE:: Fri Aug 17 09:18:04 2001
  TRACE_VER_MAJOR:: 0
  TRACE_VER_MINOR:: 97
  TRACE_LITTLE_ENDIAN:: TRUE
  TRACE_ENCODING:: 16 byte events
  TRACE_BOOT_DATE:: Fri Aug 17 09:02:16 2001
  TRACE_CYCLES_PER_SEC:: 132980400
  TRACE_CPU_NUM:: 1
  TRACE_SYSNAME:: QNX
  TRACE_NODENAME:: localhost
  TRACE_SYS_RELEASE:: 6.1.0
  TRACE_SYS_VERSION:: 2001/08/15-08:15:15
  TRACE_MACHINE:: x86pc
  TRACE_SYSPAGE_LEN:: 2248
-- KERNEL EVENTS --
.
.
.
t:0x31ea0f3d CPU:00 THREAD :THRUNNING pid:245775 tid:1
t:0x31ebec31 CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x31eca36f CPU:00 THREAD :THRUNNING pid:7 tid:5
t:0x31eea437 CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x31ef5e6d CPU:00 THREAD :THRUNNING pid:7 tid:5
t:0x31f15ab9 CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x31f21f4f CPU:00 THREAD :THRUNNING pid:7 tid:5
t:0x31f41a0b CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x31f4dfa9 CPU:00 THREAD :THRUNNING pid:7 tid:5
t:0x31f6d8a7 CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x31f79c47 CPU:00 THREAD :THRUNNING pid:7 tid:5
t:0x31f99525 CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x31fa5d5f CPU:00 THREAD :THRUNNING pid:7 tid:5
t:0x31fc59a9 CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x31fd2879 CPU:00 THREAD :THRUNNING pid:7 tid:5
t:0x31ff2137 CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x31fffe11 CPU:00 THREAD :THRUNNING pid:7 tid:5
t:0x32031b27 CPU:00 KER_CALL:RING0/02 func_p:ff836f16 arg_p:e3ff5e9c
t:0x3203584f CPU:00 THREAD :THRUNNING pid:98317 tid:1
t:0x3204df2d CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x32058db7 CPU:00 THREAD :THRUNNING pid:7 tid:5
t:0x32078fcd CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x32084dbb CPU:00 THREAD :THRUNNING pid:7 tid:5

```

```

t:0x320a471b CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x320c3501 CPU:00 THREAD :THRUNNING pid:7 tid:5
t:0x320e2ebd CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
.
.
.
t:0x35e2c6cf CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x35e38657 CPU:00 THREAD :THRUNNING pid:7 tid:5
t:0x35e59269 CPU:00 KER_CALL:RING0/02 func_p:ff82874e arg_p:0
t:0x35e6766d CPU:00 THREAD :THRUNNING pid:7 tid:5

```

### C File: `usr_event_simple.c`

```

/*
 * Copyright 2003, QNX Software Systems Ltd. All Rights Reserved.
 *
 * This source code may contain confidential information of QNX Software
 * Systems Ltd. (QSSL) and its licensors. Any use, reproduction,
 * modification, disclosure, distribution or transfer of this software,
 * or any software which includes or is based upon any of this code, is
 * prohibited unless expressly authorized by QSSL by written agreement. For
 * more information (including whether this source code file has been
 * published) please email licensing@qnx.com.
 */

#ifdef __USAGE
%C - instrumentation example

%C - example that illustrates the very basic use of
the TraceEvent() kernel call and the instrumentation
module with tracelogger in a daemon mode.

All classes and their events are included and monitored.
Additionally, four user generated simple events and
one string event are intercepted.

In order to use this example, start the tracelogger
in the daemon mode as:

tracelogger -n iter_number -dl

with iter_number = your choice of 1 through 10

After executing the example, the tracelogger (daemon)
will log the specified number of iterations and then
terminate. There are no messages printed upon successful
completion of the example. You can view the intercepted
events with the traceprinter utility. The intercepted
user events (class USREVENT) have event id(s)
(EVENT) equal to: 111, 222, 333, 444 and 555.

See accompanied documentation and comments within
the example source code for more explanations.
#endif

#include <sys/trace.h>
#include <unistd.h>

```

```
#include "instrex.h"

int main(int argc, char **argv)
{
    /*
     * Just in case, turn off all filters, since we
     * don't know their present state - go to the
     * known state of the filters.
     */
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_DELALLCLASSES));
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_KERCALL));
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_KERCALL));
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_THREAD));
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_THREAD));

    /*
     * Set fast emitting mode for all classes and
     * their events.
     */
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_SETALLCLASSESFAST));

    /*
     * Intercept all event classes
     */
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_ADDALLCLASSES));

    /*
     * Start tracing process
     *
     * During the tracing process, the tracelogger (which
     * is being executed in a daemon mode) will log all events.
     * The number of full logged iterations is user specified.
     */
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_START));

    /*
     * Insert four user defined simple events and one string
     * event into the event stream. The user events have
     * arbitrary event id(s): 111, 222, 333, 444 and 555
     * (possible values are in the range 0...1023).
     * Every user event with id=(111, ..., 555) has attached
     * two numerical data (simple event): ({1,11}, ..., {4,44})
     * and string (string event id=555) "Hello world".
     */
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_INSERTSUSEREVENT, 111, 1, 11));
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_INSERTSUSEREVENT, 222, 2, 22));
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_INSERTSUSEREVENT, 333, 3, 33));
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_INSERTSUSEREVENT, 444, 4, 44));
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_INSERTUSRSTREVENT, 555, "Hello world" ));

    /*
     * The main() of this execution flow returns.
     * However, the main() function of the tracelogger
     * will return after registering the specified number
     * of events.
     */
    return (0);
}
```

Sample result file: `usr_event_simple.results.txt`

```

TRACEPRINTER version 1.02
TRACEPARSER LIBRARY version 1.02
-- HEADER FILE INFORMATION --
  TRACE_FILE_NAME:: /dev/shmem/tracebuffer
  TRACE_DATE:: Mon Apr 25 05:37:55 1988
  TRACE_VER_MAJOR:: 1
  TRACE_VER_MINOR:: 02
  TRACE_LITTLE_ENDIAN:: TRUE
  TRACE_ENCODING:: 16 byte events
  TRACE_BOOT_DATE:: Mon Apr 25 02:28:06 1988
  TRACE_CYCLES_PER_SEC:: 400013900
  TRACE_CPU_NUM:: 2
  TRACE_SYSNAME:: QNX
  TRACE_NODENAME:: localhost
  TRACE_SYS_RELEASE:: 6.2.1
  TRACE_SYS_VERSION:: 2002/12/09-10:13:08est
  TRACE_MACHINE:: x86pc
  TRACE_SYSPAGE_LEN:: 2400
-- KERNEL EVENTS --
t:0x74a43dfa CPU:01 CONTROL :TIME msb:0x00000427, lsb(offset):0x74a43cb6
.
.
.
t:0x74a4d699 CPU:01 PROCESS :PROCCREATE_NAME
      ppid:10
      pid:28686
name:/cvs/utills/t/tracprinter/examples/usr_event_simple/nto/x86/o/examples-usr_event_simple
t:0x74a4dcb2 CPU:01 THREAD :THCREATE      pid:28686 tid:1
t:0x74a4ddb7 CPU:01 THREAD :THRUNNING     pid:28686 tid:1
t:0x74a4e0a8 CPU:01 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000 empty:0x00000000
t:0x74a4e3a2 CPU:01 KER_CALL:TRACE_EVENT/01 mode:0x4000001e class[header]:0x0000006f
t:0x74a4e4f1 CPU:01 USREVENT:EVENT:111, d0:0x00000001 d1:0x0000000b
t:0x74a4e650 CPU:01 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000 empty:0x00000000
t:0x74a4e8e5 CPU:01 KER_CALL:TRACE_EVENT/01 mode:0x4000001e class[header]:0x000000de
t:0x74a4e9dc CPU:01 USREVENT:EVENT:222, d0:0x00000002 d1:0x00000016
t:0x74a4eb51 CPU:01 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000 empty:0x00000000
t:0x74a4ee23 CPU:01 KER_CALL:TRACE_EVENT/01 mode:0x4000001e class[header]:0x0000014d
t:0x74a4ef07 CPU:01 USREVENT:EVENT:333, d0:0x00000003 d1:0x00000021
t:0x74a4f07c CPU:01 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000 empty:0x00000000
t:0x74a4f300 CPU:01 KER_CALL:TRACE_EVENT/01 mode:0x4000001e class[header]:0x000001bc
t:0x74a4f41e CPU:01 USREVENT:EVENT:444, d0:0x00000004 d1:0x0000002c
t:0x74a4f58a CPU:01 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000 empty:0x00000000
t:0x74a4f826 CPU:01 KER_CALL:TRACE_EVENT/01 mode:0x30000020 class[header]:0x0000022b
t:0x74a4fa4f CPU:01 USREVENT:EVENT:555 STR:"Hello world"
t:0x74a4fc63 CPU:01 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000 empty:0x00000000
.
.
.

```



## **Chapter 9**

---

# **Hints and Troubleshooting**

### ***In this chapter...***

How to check for the Instrumented Kernel mode    83  
Run as **root**    83  
Monitor diskspace    83



## How to check for the Instrumented Kernel mode

When in Instrumented Kernel mode, nothing is visibly different. The performance won't noticeably change either. Thus, it can be tough to tell whether you've successfully changed into Instrumented Kernel mode. And forgetting to start the Instrumented Kernel is something we all do once in a while.

To check for Instrumented Kernel mode, type:

```
ls /proc/boot
```

If one of the files listed is `procnto-*instr`, you're successfully running the Instrumented Kernel. But if the file is `procnto`, you're running the noninstrumented kernel

To start the Instrumented Kernel, see the Tutorial chapter.

## Run as root

The data-capture utilities require `root` privileges to allocate buffer memory or to use functions such as `InterruptHookTrace()`.

Data-capture utilities won't work properly without these privileges.

## Monitor diskpace

Because the `tracelogger` may write data at rates well in excess of 20 M/min, running it for prolonged periods or running it repeatedly can use up a surprisingly large amount of space. If diskpace is low, wipe old log files regularly. (In its default mode, `tracelogger` overwrites its previous default file.)



*Chapter 10*  

---

**Functions**



This chapter includes descriptions of the functions of the System Analysis Toolkit.

- *InterruptHookTrace()*
- *TraceEvent()*
- *traceparser()*
- *traceparser\_cs()*
- *traceparser\_cs\_range()*
- *traceparser\_debug()*
- *traceparser\_destroy()*
- *traceparser\_get\_info()*
- *traceparser\_init()*

## ***InterruptHookTrace()***

© 2005, QNX Software Systems

*Attach the pseudo interrupt handler that's used by the instrumented module*

### **Synopsis:**

```
#include <sys/neutrino.h>

int InterruptHookTrace(
    const struct sigevent * (* handler)(int),
    unsigned flags );
```

### **Library:**

libc

### **Description:**

The *InterruptHookTrace()* function attaches the pseudo interrupt handler *handler* which is used by the instrumented module.

Before calling this function, the thread must request I/O privileges by calling:

```
ThreadCtl( _NTO_TCTL_IO, 0 );
```

The *handler* argument specifies the pseudo interrupt handler that receives trace events from the kernel.

The *flags* argument is a bitwise OR of the following values, or 0:

<b>Flag</b>	<b>Description</b>
<code>_NTO_INTR_FLAGS_END</code>	Put the new handler at the end of the list of existing handlers (for shared interrupts) instead of the start.

### **`_NTO_INTR_FLAGS_END`**

The interrupt structure allows trace interrupts to be shared. For example, if two processes take over the same trace interrupt, both handlers are invoked consecutively. When a handler attaches, it's placed in front of any existing handlers for that interrupt and is called

first. This behavior can be changed by setting the `_NTO_INTR_FLAGS_END` flag in the *flags* argument. This adds the handler at the end of any existing handlers.

**Blocking states**

This call doesn't block.

**Returns:**

An interrupt function ID, or -1 if an error occurs (*errno* is set).

**Errors:**

EAGAIN	All kernel interrupt entries are in use.
EFAULT	A fault occurred when the kernel tried to access the buffers provided.
EPERM	The process doesn't have superuser capabilities.

**Classification:**

QNX 6

**Safety**

---

Cancellation point	No
Interrupt handler	No
Signal handler	Yes
Thread	Yes

**See also:**

*TraceEvent()*

# TraceEvent()

© 2005, QNX Software Systems

Trace kernel events

## Synopsis:

```
#include <sys/neutrino.h>
#include <sys/trace.h>

int TraceEvent( int mode,
                ... );
```

## Library:

libc

## Description:

The *TraceEvent()* function controls all stages of the instrumentation process including initialization, starting, stopping, filter control and event insertion. These stages are broadly grouped into the following categories:

- Buffer and execution control
- Fast/wide mask configuration
- Static rules filter configuration
- Dynamic rules filter configuration
- User-generated trace events

This description also includes these sections:

- Argument descriptions
- *class* argument descriptions



---

Filter and mask settings are made regardless of the previous settings. Use care to not accidentally override or delete previous configurations.

---

## Buffer and execution control

These modes control the buffer set up as well as start and stop logging.

### **\_NTO\_TRACE\_ALLOCBUFFER, \_NTO\_TRACE\_DEALLOCBUFFER**

```
TraceEvent(_NTO_TRACE_ALLOCBUFFER, uint bufnum, void** linkliststart)
TraceEvent(_NTO_TRACE_DEALLOCBUFFER)
```

The allocation option creates and initializes the internal circular link list of trace buffers. The created and initialized trace buffers hold the emitting trace events.

<i>bufnum</i>	Number of buffers to allocate.
<i>*linkliststart</i>	Physical address of the beginning of the circular link list of allocated trace buffers.

Allocated trace buffers can store 1024 simple trace events.




---

If your application calls this mode, it must run as **root**.

---

The deallocation option deallocates all of the previously allocated trace buffers. All events stored inside of the trace buffers are lost.

### **\_NTO\_TRACE\_FLUSHBUFFER**

```
TraceEvent(_NTO_TRACE_FLUSHBUFFER)
```

Forces flushing of the buffer regardless of the trace event number it contains.

### **\_NTO\_TRACE\_QUERYEVENTS**

```
TraceEvent(_NTO_TRACE_QUERYEVENTS)
```

Returns the number of simple trace events that's currently stored in the trace buffer.

## **\_NTO\_TRACE\_START, \_NTO\_TRACE\_STARTNOSTATE, \_NTO\_TRACE\_STOP**

```
TraceEvent( _NTO_TRACE_START )  
TraceEvent( _NTO_TRACE_STARTNOSTATE )  
TraceEvent( _NTO_TRACE_STOP )
```

Starts/stops the instrumentation process. The event stream containing the trace events is opened/closed.

The `_NTO_TRACE_START` and `_NTO_TRACE_STARTNOSTATE` options are the same except the latter suppresses the initial system state information (names of processes and thread IDs.)

## **\_NTO\_TRACE\_SETRINGMODE**

```
TraceEvent( _NTO_TRACE_SETRINGMODE )
```

Sets a ring mode of the internal circular link list. When an external application uses this mode, the kernel stores all events in a circular fashion inside the link list without flushing them. The maximum capturing time (without history overwriting) is determined by the number of allocated buffers, as well as by the number of generated trace events.

## **\_NTO\_TRACE\_SETLINEARMODE**

```
TraceEvent( _NTO_TRACE_SETLINEARMODE )
```

Sets a default linear mode of the internal circular link list. When using this mode, every filled-up buffer is captured and flushed immediately.

## **Fast/wide mask configuration**

These modes control the operation of the fast/wide mask. For more information about this mask, see the Filtering chapter in this guide.




---

Currently, only the kernel call related classes are affected by the fast/wide modes. In fast mode, only two of the most important kernel call arguments and two of the most important kernel call return values are intercepted. See the Kernel call arguments and return values appendix for the list of the intercepted arguments and return values.

---

### **\_NTO\_TRACE\_SETALLCLASSESFAST, \_NTO\_TRACE\_SETALLCLASSESWIDE**

```
TraceEvent(_NTO_TRACE_SETALLCLASSESFAST)
TraceEvent(_NTO_TRACE_SETALLCLASSESWIDE)
```

Sets the fast/wide emitting mode for all classes and events.

### **\_NTO\_TRACE\_SETCLASSFAST, \_NTO\_TRACE\_SETCLASSWIDE**

```
TraceEvent(_NTO_TRACE_SETCLASSFAST, int class)
TraceEvent(_NTO_TRACE_SETCLASSWIDE, int class)
```

Sets the fast/wide emitting mode for all events within the specified *class*.

### **\_NTO\_TRACE\_SETEVENTFAST, \_NTO\_TRACE\_SETEVENTWIDE**

```
TraceEvent(_NTO_TRACE_SETEVENTFAST, int class, int event)
TraceEvent(_NTO_TRACE_SETEVENTWIDE, int class, int event)
```

Sets the fast/wide emitting mode for the specified *event* for the specified *class*.

### **Static rules filter configuration**

These modes control the operation of the static rules filter. For more information about this filter, see the Filtering chapter in this guide.

### **\_NTO\_TRACE\_ADDALLCLASSES, \_NTO\_TRACE\_DELALLCLASSES**

```
TraceEvent(_NTO_TRACE_ADDALLCLASSES)
TraceEvent(_NTO_TRACE_DELALLCLASSES)
```

Emit/suppress trace events for all classes and events.

## **\_NTO\_TRACE\_ADDCLASS, \_NTO\_TRACE\_DELCLASS**

```
TraceEvent(_NTO_TRACE_ADDCLASS, class)
TraceEvent(_NTO_TRACE_DELCLASS, class)
```

Emit/suppress all trace events from a specific *class*.

## **\_NTO\_TRACE\_ADDEVENT, \_NTO\_TRACE\_DELEVENT**

```
TraceEvent(_NTO_TRACE_ADDEVENT, class, event)
TraceEvent(_NTO_TRACE_DELEVENT, class, event)
```

Emit/suppress a trace *event* from a specific *class*.

## **\_NTO\_TRACE\_SETCLASSPID, \_NTO\_TRACE\_CLRCLASSPID, \_NTO\_TRACE\_SETCLASSTID, \_NTO\_TRACE\_CLRCLASSTID**

```
TraceEvent(_NTO_TRACE_SETCLASSPID, int class, pid_t pid)
TraceEvent(_NTO_TRACE_CLRCLASSPID, int class)
TraceEvent(_NTO_TRACE_SETCLASSTID, int class, pid_t pid, tid_t tid)
TraceEvent(_NTO_TRACE_CLRCLASSTID, int class)
```

Emits/suppresses all events from a specified process ID (and thread ID).

## **\_NTO\_TRACE\_SETEVENTPID, \_NTO\_TRACE\_CLREVENTPID, \_NTO\_TRACE\_SETEVENTTID, \_NTO\_TRACE\_CLREVENTTID,**

```
TraceEvent(_NTO_TRACE_SETEVENTPID, int class, int event, pid_t pid)
TraceEvent(_NTO_TRACE_CLREVENTPID, int class, int event)
TraceEvent(_NTO_TRACE_SETEVENTTID, int class, int event, pid_t pid, tid_t tid)
TraceEvent(_NTO_TRACE_CLREVENTTID, int class, int event)
```

Emits/suppresses a specific *event* for a specified process ID (and thread ID.)

## Dynamic rules filter configuration

These modes control the operation of the dynamic rules filter. For more information about this filter, see the Filtering chapter in this guide.

## Event Handler Data Access

The access to the trace event information from within the event handler can be done using members of the `event_data_t`.

The valid layout of the `event_data_t` structure (declared in `sys/trace.h`) is as follow:

```
/* event data filled by an event handler */
typedef struct
{
    __traceentry header;          /* same as traceevent header */
    _Uint32t* data_array;        /* initialized by the user */
    _Uint32t el_num;             /* number of elements returned */
    void* area;                  /* user data */
    _Uint32t feature_mask;       /* bits indicate valid features */
    _Uint32t feature[_NTO_TRACE_FI_NUM]; /* feature array
                                     - additional data */
} event_data_t;
```

The bits of the member `feature_mask` are related to any additional feature (argument) that could be accessed inside the event handler. All standard data arguments, the ones that correspond to the data arguments of the trace-event, are delivered without changes within array `data_array[]`. If any particular bit of the `feature_mask` is set to value equal to 1, then, the feature corresponding to this bit can be accessed within array `feature[]`. Otherwise, the feature should not be accessed. For example, if the expression:

```
feature_mask & _NTO_TRACE_FMPID
```

has its logical value equal to `TRUE`, then, the additional feature corresponding to identifier `_NTO_TRACE_FMPID` (PID) can be accessed as:

```
my_pid = feature[_NTO_TRACE_FIPID];
```

For every additional feature there have been provided two constants:

**\_NTO\_TRACE\_FM\*\*\*** - feature parameter masks

**\_NTO\_TRACE\_FI\*\*\*** - feature index parameters

to check and to access the given feature.

## **\_NTO\_TRACE\_ADDEVENTHANDLER, \_NTO\_TRACE\_DELEVENTHANDLER**

```
TraceEvent(_NTO_TRACE_ADDEVENTHANDLER,  
           class,  
           event,  
           int (*event_hdlr)(event_data_t*),  
           event_data_t* data_struct)  
TraceEvent(_NTO_TRACE_DELEVENTHANDLER,  
           class,  
           event)
```

## **\_NTO\_TRACE\_ADDCLASSEVHANDLER, \_NTO\_TRACE\_DELCLASSEVHANDLER**

```
TraceEvent(_NTO_TRACE_ADDCLASSEVHANDLER,  
           class,  
           int (*event_hdlr)(event_data_t*),  
           event_data_t* data_struct)  
TraceEvent(_NTO_TRACE_DELCLASSEVHANDLER,  
           class)
```

Attaches/deletes the event handler for a specified class, where:

*event\_hdlr*      Pointer to the event handler.

*data\_struct*    Pointer to the data structure **event\_data\_t**.

In order to emit an event data, a dynamic filter (event handler) has to return 1. If both types of the dynamic filters (event handler and class event handler) are applicable to a particular event, the event is emitted if both event handlers return 1.

**User-generated trace events**

These modes control the insertion of “fake” events into the event stream.

**\_NTO\_TRACE\_INSERTEVENT**

```
TraceEvent(_NTO_TRACE_INSERTEVENT, int head, int stamp, int data0, int data1)
```

Inserts a generic, “real” event into the event stream. It’s powerful but because the API doesn’t do any of the interpretation for you, this function should be used with care by advanced users only. The data-interpretation program must be modified to properly interpret the event.

The arguments are:

*head*      Header of the trace event.

*stamp*     Time stamp.

*data0*     Data d0.

*data1*     Data d1.

**\_NTO\_TRACE\_INSERTSUSEREVENT, \_NTO\_TRACE\_INSERTCUSEREVENT, \_NTO\_TRACE\_INSERTUSRSTREVENT**

```
TraceEvent(_NTO_TRACE_INSERTSUSEREVENT, int event, int data0, int data1)
```

```
TraceEvent(_NTO_TRACE_INSERTCUSEREVENT, int event, unsigned * buf, unsigned len)
```

```
TraceEvent(_NTO_TRACE_INSERTUSRSTREVENT, int event, const char * str)
```




---

The *len* argument represents the number of integers in *buf*.

---

These modes insert user-created events into the event stream. Because the API handles details such as timestamping, they’re reasonably easy to use.

**\_NTO\_TRACE\_INSERTSUSEREVENT**

Simple user event.

`_NTO_TRACE_INSERTUSEREVENT`

Complex user event.

`_NTO_TRACE_INSERTUSRSTREVENT`

User string event.

The arguments are:

*event* User defined event code. The value should be between `_NTO_TRACE_USERFIRST` and `_NTO_TRACE_USERLAST`.

*str* Null terminated string.

*data0* Data d0.

*data1* Data d1.

The *TraceEvent()* function controls all stages of the instrumentation process such as initialization, starting, execution control, and stopping. These stages consist of the following activities:

- creating an internal circular linked list of trace buffers
- initializing filters
- turning on or off the event stream
- deallocating the internal circular linked list of trace buffers

The *TraceEvent()* function accepts any number of arguments grouped logically as follows:

```
TraceEvent(mode [,class [,event]] [,p1 [,p2 [,p3 ... [,pn]]]])
```

Here's a description of the arguments:

***mode***

Specifies a control action (compulsory).

You'll find a description for each *mode* argument listed below in the *mode* argument descriptions section.

Some *mode* arguments require additional arguments; see the table of argument hierarchy for details. Valid arguments are:

```
_NTO_TRACE_ADDALLCLASSES
_NTO_TRACE_ADDCLASS
_NTO_TRACE_ADDEVENT
_NTO_TRACE_ADDEVENTHANDLER
_NTO_TRACE_ALLOCBUFFER
_NTO_TRACE_CLRCLASSPID
_NTO_TRACE_CLRCLASSTID
_NTO_TRACE_CLREVENTPID
_NTO_TRACE_CLREVENTTID
_NTO_TRACE_DEALLOCBUFFER
_NTO_TRACE_DELALLCLASSES
_NTO_TRACE_DELCLASS
_NTO_TRACE_DELEVENT
_NTO_TRACE_DELEVENTHANDLER
_NTO_TRACE_FLUSHBUFFER
_NTO_TRACE_INSERTCUSEREVENT
_NTO_TRACE_INSERTEVENT
_NTO_TRACE_INSERTUSEREVENT
_NTO_TRACE_INSERTUSRSTREVENT
_NTO_TRACE_QUERYEVENTS
_NTO_TRACE_SETRINGMODE
_NTO_TRACE_SETLINEARMODE
_NTO_TRACE_SETALLCLASSESFAST
_NTO_TRACE_SETALLCLASSESWIDE
_NTO_TRACE_SETCLASSFAST
```

.\_NTO\_TRACE\_SETCLASSPID  
.\_NTO\_TRACE\_SETCLASSTID  
.\_NTO\_TRACE\_SETCLASSWIDE  
.\_NTO\_TRACE\_SETEVENTFAST  
.\_NTO\_TRACE\_SETEVENTPID  
.\_NTO\_TRACE\_SETEVENTTID  
.\_NTO\_TRACE\_SETEVENTWIDE  
.\_NTO\_TRACE\_START  
.\_NTO\_TRACE\_STOP  
.\_NTO\_TRACE\_STARTNOSTATE

## *class*

You'll find a description for each *class* argument listed below in the *class* argument descriptions section. Some *class* arguments may require additional arguments; see the table of argument hierarchy for details.

Valid arguments are:

.\_NTO\_TRACE\_CONTROL  
.\_NTO\_TRACE\_INT  
.\_NTO\_TRACE\_INTENTER  
.\_NTO\_TRACE\_INTEXT  
.\_NTO\_TRACE\_KERCALL  
.\_NTO\_TRACE\_KERCALLEENTER  
.\_NTO\_TRACE\_KERCALLEEXIT  
.\_NTO\_TRACE\_PROCESS  
.\_NTO\_TRACE\_THREAD  
.\_NTO\_TRACE\_VTHREAD

***event***

Redirects the control action specified by the *mode* and *class* towards a trace event within the class.

You'll find a description for each *event* argument listed below in the *event* argument descriptions section. Some *event* arguments may require additional arguments; see the table of argument hierarchy for details.

The following table shows the valid *event* arguments for a particular *class*:

<b>If the value of <i>class</i> is:</b>	<b>Then a valid <i>event</i> argument is:</b>
_NTO_TRACE_CONTROL	_NTO_TRACE_CONTROLTIME
_NTO_TRACE_INT, _NTO_TRACE_INTENTER, _NTO_TRACE_INTEXTIT	a logical interrupt vector number
_NTO_TRACE_KERCALL, _NTO_TRACE_KERCALLEENTER, _NTO_TRACE_KERCALLEEXIT	A valid <code>__KER_*</code> keyword from <code>&lt;sys/kercalls.h&gt;</code> (such as <code>__KER_MSG_SENDDV</code> .)
_NTO_TRACE_PROCESS	_NTO_TRACE_PROCCREATE, _NTO_TRACE_PROCCREATE_NAME, _NTO_TRACE_PROCDESTROY, _NTO_TRACE_PROCDESTROY_NAME

*continued...*

---

**If the value of *class* is:**

\_NTO\_TRACE\_THREAD

---

**Then a valid *event* argument is:**

\_NTO\_TRACE\_THCONDVAR,  
\_NTO\_TRACE\_THCREATE,  
\_NTO\_TRACE\_THDEAD,  
\_NTO\_TRACE\_THDESTROY,  
\_NTO\_TRACE\_THINTR,  
\_NTO\_TRACE\_THJOIN,  
\_NTO\_TRACE\_THMUTEX,  
\_NTO\_TRACE\_THNANOSLEEP,  
\_NTO\_TRACE\_THNET\_REPLY,  
\_NTO\_TRACE\_THNET\_SEND,  
\_NTO\_TRACE\_THREADY,  
\_NTO\_TRACE\_THRECEIVE,  
\_NTO\_TRACE\_THREPLY,  
\_NTO\_TRACE\_THRUNNING,  
\_NTO\_TRACE\_THSEM,  
\_NTO\_TRACE\_THSEND,  
\_NTO\_TRACE\_THSIGSPEND,  
\_NTO\_TRACE\_THSIGWAITINFO,  
\_NTO\_TRACE\_THSTACK,  
\_NTO\_TRACE\_THSTOPPED,  
\_NTO\_TRACE\_THWAITCTX,  
\_NTO\_TRACE\_THWAITPAGE,  
\_NTO\_TRACE\_THWAITTHREAD

*continued...*

**If the value of *class* is:**`_NTO_TRACE_VTHREAD`**Then a valid *event* argument is:**

```

_NTO_TRACE_VTHCONDVAR,
_NTO_TRACE_VTHCREATE,
_NTO_TRACE_VTHDEAD,
_NTO_TRACE_VTHDESTROY,
_NTO_TRACE_VTHINTR,
_NTO_TRACE_VTHJOIN,
_NTO_TRACE_VTHMUTEX,
_NTO_TRACE_VTHNANOSLEEP,
_NTO_TRACE_VTHNET_REPLY,
_NTO_TRACE_VTHNET_SEND,
_NTO_TRACE_VTHREADY,
_NTO_TRACE_VTHRECEIVE,
_NTO_TRACE_VTHREPLY,
_NTO_TRACE_VTHRUNNING,
_NTO_TRACE_VTHSEM,
_NTO_TRACE_VTHSEND,
_NTO_TRACE_VTHSIGSUSPEND,
_NTO_TRACE_VTHSIGWAITINFO,
_NTO_TRACE_VTHSTACK,
_NTO_TRACE_VTHSTOPPED,
_NTO_TRACE_VTHWAITCTX,
_NTO_TRACE_VTHWAITPAGE,
_NTO_TRACE_VTHWAITTHREAD

```

***p1...pn***

Specifies any additional parameters that are required to perform the desired control action.

**Argument descriptions**

The following are the generic arguments used for the *TraceEvent()* modes. Mode-specific arguments accompany the mode description:

*mode*      The control action. The *mode* is always the first argument in the *TraceEvent()* function. Depending upon the value of *mode*, further arguments may be necessary. The

description of what each *mode* does appears earlier in this section. Examples of the *mode* include:

`_NTO_TRACE_ALLOCBUFFER,`  
`_NTO_TRACE_SETCLASSFAST.`

*class* The category of events. All the events are logically grouped into several classes. A list of valid classes is given in *class* argument descriptions, later in this section.

*event* The event. Because the events are grouped by *class*, the *event* must be a member of the *class* in order to be valid. A list of events can be found in the Kernel Call Arguments and Return Values chapter in this guide.

*pid* Process ID to be registered.

*tid* Thread ID to be registered.

## ***class* argument descriptions**

The *class* argument may be one of the following:

`_NTO_TRACE_CONTROL`

Specifies the set of control events (i.e. time-overflow event) that're used by the communication protocol between the microkernel and `tracelogger`.

`_NTO_TRACE_INTENTER,`  
`_NTO_TRACE_INTEXTIT`

Specifies the set of interrupt entry/exit events.

`_NTO_TRACE_KERCALLEENTER,`  
`_NTO_TRACE_KERCALLEEXIT`

Specifies the set of kernel call entry/exit events.

`_NTO_TRACE_PROCESS`

Specifies the set of events associated with process creation and destruction.

`_NTO_TRACE_THREAD`,  
`_NTO_TRACE_VTHREAD`

Specifies the set of *class* arguments that contain thread (or virtual thread) state changes, and create or destroy events.

There are also “pseudo classes” offered as a convenience:

`_NTO_TRACE_KERCALL`

Specifies all of the kernel call events:

`_NTO_TRACE_KERCALLEENTER` and  
`_NTO_TRACE_KERCALLEEXIT`.

`_NTO_TRACE_INT`

Specifies all of the interrupt events: `_NTO_TRACE_INTENTER`  
and `_NTO_TRACE_INTEXIT`.

## Returns:

If *mode* is set to `_NTO_TRACE_QUERYEVENTS`

Number of events in the buffer, or -1 if an error occurs (*errno* is set).

If *mode* isn't set to `_NTO_TRACE_QUERYEVENTS`

0 for success, or -1 if an error occurs (*errno* is set).

## Errors:

<code>ECANCELED</code>	The requested action has been canceled.
<code>EFAULT</code>	Bad internal trace buffer address. The requested action has been specified out of order.
<code>ENOMEM</code>	Insufficient memory to allocate the trace buffers.
<code>ENOSUP</code>	The requested action isn't supported.
<code>EPERM</code>	The application doesn't have permission to perform the action.

## Classification:

QNX 6

### Safety

---

Cancellation point	No
Interrupt handler	Read the <i>Caveats</i>
Signal handler	Yes
Thread	Yes

## Caveats:

You can call *TraceEvent()* from an interrupt/event handler. However, not all trace modes are valid in this case. The valid trace modes are:

- `_NTO_TRACE_INSERTSUSEREVENT`
- `_NTO_TRACE_INSERTCUSEREVENT`
- `_NTO_TRACE_INSERTUSRSTREVENT`
- `_NTO_TRACE_INSERTEVENT`
- `_NTO_TRACE_STOP`
- `_NTO_TRACE_STARTNOSTATE`
- `_NTO_TRACE_START`

## See also:

*InterruptAttach()*, *InterruptHookTrace()*

**Synopsis:**

```
#include <sys/traceparser.h>

extern int traceparser (
    struct traceparser_state * stateptr,
    void * userdata,
    const char * filename );
```

**Library:**

libtraceparser

**Description:**

The *traceparser()* function starts the parsing procedure *filename*. It also executes the user defined callback functions and passes the *userdata* to it. The *stateptr* argument is an opaque structure obtained from *traceparser\_init()*.

**Returns:**

0      Success

-1     Failure; *errno* is set. See also *traceparser\_get\_info()* for further details.

**Classification:**

QNX 6

**Safety**

---

Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

**See also:**

*traceparser\_get\_info()*, *traceparser\_init()*

**Synopsis:**

```
#include <traceparser.h>

int traceparser_cs (
    struct traceparser_state * stateptr,
    void * userdata,
    tracep_callb_func_t funcptr,
    unsigned class,
    unsigned event );
```

**Library:**

libtraceparser

**Description:**

The *traceparser\_cs()* function attaches one callback function, specified by the pointer *funcptr*, to one particular *event*, from one particular *class*. The user data (*userdata*) is passed to the attached callback function upon execution. The *stateptr* is an opaque structure obtained from *traceparser\_init()*.

**Returns:**

- 0 Success; a pointer to the event
- 1 Failure; *errno* is set. See also *traceparser\_get\_info()* for further details.

**Classification:**

QNX 6

**Safety**

---

Cancellation point No

Interrupt handler No

*continued...*

**Safety**

---

Signal handler	No
Thread	No

**See also:**

*traceparser\_get\_info()*, *traceparser\_init()*

**Synopsis:**

```
#include <traceparser.h>

int traceparser_cs_range (
    struct traceparser_state * stateptr,
    void * userdata,
    tracep_callb_func_t funcptr,
    unsigned class,
    unsigned firstevent,
    unsigned lastevent );
```

**Library:**

libtraceparser

**Description:**

The *traceparser\_cs\_range()* function attaches one callback function, given by the pointer *funcptr*, to a range of events from *firstevent* through to *lastevent* inclusive, from one particular *class*. The user data (*userdata*) is passed to the registered callback function (*funcptr*) upon execution. The *stateptr* is an opaque structure obtained from *traceparser\_init()*.

**Returns:**

- 0 Success; a pointer to the list of events.
- 1 Failure; *errno* is set. See also *traceparser\_get\_info()* for further details

**Classification:**

QNX 6

**Safety**

---

Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

**See also:**

*traceparser\_get\_info()*, *traceparser\_init()*

**Synopsis:**

```
#include <traceparser.h>

int traceparser_debug (
    struct traceparser_state * stateptr,
    FILE * streamptr,
    unsigned flags );
```

**Library:**

libtraceparser

**Description:**

The *traceparser\_debug()* function sets the debug modes of the traceparser module. The *streamptr* argument is a pointer to the debug output stream; *flags* specifies the debug category. The *stateptr* is an opaque structure obtained from *traceparser\_init()*.

**Debug flags**

The following is a list of the arguments that may be used for *flags* and the debug level for each:

`_TRACEPARSER_DEBUG_ALL`

Everything.

`_TRACEPARSER_DEBUG_ERRORS`

Critical errors only.

`_TRACEPARSER_DEBUG_EVENTS`

Row input events only.

`_TRACEPARSER_DEBUG_HEADER`

Header information only.

`_TRACEPARSER_DEBUG_NONE`

No debugging.

`_TRACEPARSER_DEBUG_SYSPAGE`

Syspage data only.

**Returns:**

- > 0     Success; a pointer to the event
- 1     Failure; *errno* is set. See also *traceparser\_get\_info()* for further details.

**Classification:**

QNX 6

**Safety**

---

Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	Yes

**See also:**

*traceparser\_get\_info()*, *traceparser\_init()*

**Synopsis:**

```
#include <traceparser.h>

void traceparser_destroy (
    struct traceparser_state ** stateptr );
```

**Library:**

libtraceparser

**Description:**

The *traceparser\_destroy()* function destroys a previously initialized traceparser state structure, *stateptr*.

**Classification:**

QNX 6

**Safety**

---

Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

**See also:**

*traceparser\_get\_info()*, *traceparser\_init()*

## ***traceparser\_get\_info()***

© 2005, QNX Software Systems

*Get information about traceparser*

### **Synopsis:**

```
#include <sys/traceparser.h>

void * traceparser_get_info (
    struct traceparser_state * stateptr,
    info_modes_t infomodes,
    unsigned * len );
```

### **Library:**

libtraceparser

### **Description:**

The *traceparser\_get\_info()* function gets information related to the state of the traceparser. The *infomodes* argument may be any of the constants shown below and are defined in **traceparser.h**.

The *len* argument is a pointer to the size of the return buffer. When specified, its contents are changed to indicate the size of the return. This is primarily for the `TRACEPARSER_SYSPAGE` and `TRACEPARSER_HEADER_KEYWORDS` modes but it'll work for all the modes. For most of the modes, *len* may be NULL.

The *stateptr* is an opaque structure obtained from *traceparser\_init()*.

### **User info modes for *info\_modes***

The following are valid user info modes; see the list below for others.

<b>Value for <i>info_modes</i> and Pointer to return data type, cast as void</b>	<b>Description</b>
<code>TRACEPARSER_INFO_SYSPAGE</code> <code>syspage_entry</code>	Returns a pointer to the syspage entry.

*continued...*

Value for <i>info_modes</i> and Pointer to return data type, cast as <code>void</code>	Description
<code>._TRACEPARSER_INFO_ENDIAN_CONV</code> <code>unsigned</code>	Returns a dereferenced pointer; 1 if the endian conversion has been applied, 0 if no conversion has been performed.
<code>._TRACEPARSER_INFO_NOW_CALLBACK_CLASS</code> <code>._TRACEPARSER_INFO_NOW_CALLBACK_EVENT</code> <code>unsigned</code>	Returns the class or event numerical value of the currently executed callback function. The numerical values are considered opaque and should be used only for other traceparser functions.
<code>._TRACEPARSER_INFO_PREV_CALLBACK_CLASS</code> <code>._TRACEPARSER_INFO_PREV_CALLBACK_EVENT</code> <code>unsigned</code>	As above, but returns the class or event numerical value of the previously executed callback function.
<code>._TRACEPARSER_INFO_PREV_CALLBACK_RETURN</code> <code>int</code>	Returns the value of the previously executed callback function.
<code>._TRACEPARSER_INFO_DEBUG</code> <code>unsigned</code>	Returns the debug category.
<code>._TRACEPARSER_INFO_ERROR</code> <code>unsigned</code>	Returns the traceparser error level. It must be used to determine traceparser library related errors. (See <code>sys/traceparser.h</code> for a list of the returned error values.)

### Other valid user info modes

The following modes return a pointer to the header field of the buffer. All data types are `void`.

- `TRACEPARSER_INFO_FILE_NAME`
- `TRACEPARSER_INFO_DATE`
- `TRACEPARSER_INFO_VER_MAJOR`
- `TRACEPARSER_INFO_VER_MINOR`

- TRACEPARSER\_INFO\_LITTLE\_ENDIAN
- TRACEPARSER\_INFO\_BIG\_ENDIAN
- TRACEPARSER\_INFO\_MIDDLE\_ENDIAN
- TRACEPARSER\_INFO\_ENCODING
- TRACEPARSER\_INFO\_BOOT\_DATE
- TRACEPARSER\_INFO\_CYCLES\_PER\_SEC
- TRACEPARSER\_INFO\_CPU\_NUM
- TRACEPARSER\_INFO\_SYSNAME
- TRACEPARSER\_INFO\_NODENAME
- TRACEPARSER\_INFO\_SYS\_RELEASE
- TRACEPARSER\_INFO\_SYS\_VERSION
- TRACEPARSER\_INFO\_MACHINE
- TRACEPARSER\_INFO\_SYSPAGE\_LEN

**Returns:**

A pointer to **void**

Success.

Null Failure; *errno* is set. See also the `_TRACEPARSER_ERROR` section of this function for further details.

**Classification:**

QNX 6

**Safety**

Cancellation point No

*continued...*

**Safety**

---

Interrupt handler	No
Signal handler	No
Thread	No

**See also:**

*traceparser\_get\_info()*, *traceparser\_init()*

## ***traceparser\_init()***

© 2005, QNX Software Systems

*Initialize or get the state of the traceparser library*

### **Synopsis:**

```
#include <sys/traceparser.h>

struct traceparser_state * traceparser_init (
    struct traceparser_state * stateptr );
```

### **Library:**

libtraceparser

### **Description:**

The *traceparser\_init()* function initializes the state of the traceparser library. To initialize the library, execute the function with the *stateptr* argument as null; the function returns the initialized state structure.

The **traceparser\_state** structure is an opaque structure for use by the other SAT functions.

### **Returns:**

A pointer to a valid initialized state structure

Success.

NULL Failure; *errno* is set. See also *traceparser\_get\_info()* for further details.

### **Classification:**

QNX 6

#### **Safety**

---

Cancellation point	No
Interrupt handler	No
Signal handler	No
Thread	No

**See also:**

*traceparser\_get\_info(), traceparser\_init()*



***Chapter 11***

---

**Utilities**



This chapter includes descriptions of the utilities related to the System Analysis Toolkit.

- **tracelogger**
- **traceprinter**

# tracelogger

© 2005, QNX Software Systems

*Logs tracing information*

## Syntax:



You must be **root** to run this utility.

```
tracelogger -w [-b num] [-d mode]
              [-f file] [-F 0-6]
              [-k num] [-M] [-n num]
              [-s num] [-S size] [-v[v...]]
```

## Runs on:

This utility runs on x86, PPC, SH-4, ARM, and MIPS processors.

## Options:

- b num** Maximum number of dynamic buffers allocated in **tracelogger**. Default is 64. Each buffer has a size of approximately 11 KB.
- d mode** Currently, the *mode* is:
  - 1 Launch in daemon mode.
- f file** Name of the file to store logged events. Default is `/dev/shmem/tracebuffer.kev`.
- F num** Filtering for different *num* values, as follows:
  - 0 Don't set any filtering.
  - 1 Disable kernel calls class.
  - 2 Disable interrupt class.
  - 3 Disable process class.
  - 4 Disable thread class.
  - 5 Disable VThread class.
  - 6 Disable COMM Class.

- k** *num*      Number of allocated buffers in the kernel. Default is 32. Each buffer has a size of approximately 16 KB.
- M**            Map the log file directly instead of writing. If you use this option, the log file must be in shared memory, and you must use the **-s** option to specify the maximum file size.
- n** *num*      Number of logged iterations. Default is 32.
- r**            Set the kernel buffer to ring mode. Default is linear mode.
- s** *num*      Number of seconds to log. The default is 0.
- s** *size*      The maximum size of the log file. Use **M** for megabytes or **K** for kilobytes. If you don't use **M** or **K**, the units are assumed to be bytes.
- v**[*v...*]    Be verbose; more **v** characters cause more verbosity.
- w**            Wide events. Default is fast events.

## Description:

The **tracelogger** utility logs the requested number of iterations, then terminates. It may be started using:

### Full tracelogger support (limited control)

Instrumentation with full tracelogger support is the simplest way to obtain the tracing information. In this case, the tracelogger performs all initialization and runtime control of the instrumentation module in addition to its normal task of saving the trace buffers to the filesystem.

### Partial tracelogger support (daemon mode, full control)

Instrumentation with partial tracelogger support is the most flexible way of controlling the instrumentation process. However, it requires the largest amount of user support during

initialization. Except for specifying the number of buffers, the `tracelogger` doesn't perform any initialization of the instrumentation module and its almost exclusive task is to log the "received" trace events to the filesystem.

## Examples:

Start `tracelogger` in wide mode and display operational information on the console screen; store the logged data in the named file and stop logging when 12 trace buffers are full.

```
#tracelogger -f /dev/shmem/my_tracebuffer -n 12 -w
```

Start `tracelogger` in *ring* mode (background) using 5 internal buffers and wait for an asynchronous signal (e.g. Ctrl-C) to stop it.

```
#tracelogger -r -b 5
```

## Exit status:

-1 An error occurred.

## Errors:

Severe errors cause `tracelogger` to terminate.

## Caveats:

Run only one instance of `tracelogger` at a time.

## See also:

`traceprinter`

**Syntax:**

```
traceprinter [-f file] [-n] [-v]
```

**Runs on:**

This utility runs on x86, PPC, SH-4, ARM, and MIPS processors and on Windows and Solaris hosts.

**Options:**

- f *file***    The filename of the file that stores the trace information to be displayed. The default is `/dev/shmem/tracebuffer`.
- n**            Remove newline characters from printed argument lines.
- v**            Display detailed information.

**Description:**

The **traceprinter** utility displays the contents of a trace file generated by **tracelogger**. The utility parses the linearly stored time events stored in the named trace file and sends the resulting formatted stream to standard output.

The formatted stream looks like this:

```
t:clk_time CPU:cpu [class: event: [P1: [P2: [P3: ... [Pn:]]]]]
```

The stream always shows the *clk\_time* and *cpu* variables; the variables in brackets are optional.

*clk\_time*        The *clk\_time* variable gives the time offset in CPU clock cycles when the trace event was registered. The 64-bit variable is broken into two 32-bit hexadecimal numbers (*msb* and *lsb*). Apart from at the start of the trace and when the *lsb* rolls over, only the *lsb* number is shown.

*cpu* The *cpu* variable is a 2-digit decimal number representing the CPU which registered the event. The variable is always 00 unless you have more than one CPU in your SMP system: if you have four CPUs, the CPU numbers range from 00 to 03. The CPU numbers are assigned to particular CPUs during initialization, when the startup programs probe the system (see **procnto\***).

Optional variables

The other variables are optional, depending on the tracing information logged. The *class* and *event* variables are followed by parameter variables, the number and type of parameters depend on the associated *class/event* pair and whether **tracelogger** used the fast or wide emitting mode. Each trace line contains one *class*, one or no *event*, and one or more parameter variables. The table below explains which parameters are shown for each combination of *class* and *event*.

one of: <i>class</i>	one of: <i>event</i>	parameters
CONTROL	TIME	P1 - msb P2 - lsb (offset)
INT_ENTR INT_EXIT	interrupt number in hexadecimal (and decimal) notation	P1 - inkernel

*continued...*

---

<b>one of: class</b>	<b>one of: event</b>	<b>parameters</b>
<b>THREAD</b>	THDEAD	P1 - PID
	THRUNNING	P2 - thread ID
	THREADY	
	THSTOPPED	
	THSEND	
	THRECEIVE	
	THREPLY	
	THSTACK	
	THWAITTHREAD	
	THWAITPAGE	
	THSIGSUSPEND	
	THSIGWAITINFO	
	THNANOSLEEP	
	THMUTEX	
	THCONDVAR	
	THJOIN	
	THINTR	
	THSEM	
	THWAITCTXN	
	THNET_SEND	
	THNET_REPLY	
	THCREATE	
	THDESTROY	

*continued...*

<b>one of: class</b>	<b>one of: event</b>	<b>parameters</b>
<b>VTHREAD</b>	VTHDEAD VTHRUNNING VTHREADY VTHSTOPPED VTHSEND VTHRECEIVE VTHREPLY VTHSTACK VTHWAITVTHREAD VTHWAITPAGE VTHSIGSUSPEND VTHSIGWAITINFO VTHNANOSLEEP VTHMUTEX VTHCONDVAR VTHJOIN VTHINTR VTHSEM VTHWAITCTXN VTHNET_SEND VTHNET_REPLY VTHCREATE VTHDESTROY	P1 - PID P2 - vthread ID
<b>PROCESS</b>	PROCCREATE PROCCREATE_NAME PROCDESTROY PROCDESTROY_NAME	P1 - parent PID P2 - PID P3 - process name (optional)
<b>KER_CALL</b>	kernel call name	P1 - kernel call number P2, P3, ... Pn - kernel call arguments
<b>KER_EXIT</b>	kernel call name	P1 - kernel call number P2, P3, ... Pn - kernel call return values

The number of kernel call arguments and return values depends on the kernel call and whether the trace file was produced using fast or wide tracing mode.

### **Examples:**

Here is an example of the first few lines of a **traceprinter** output:

```

TRACEPRINTER version 0.94
-- HEADER FILE INFORMATION --
    TRACE_FILE_NAME:: /scratch/quadlog
    TRACE_DATE:: Fri Jun  8 13:14:40 2001
    TRACE_VER_MAJOR:: 0
    TRACE_VER_MINOR:: 96
    TRACE_LITTLE_ENDIAN:: TRUE
    TRACE_ENCODING:: 16 byte events
    TRACE_BOOT_DATE:: Fri Jun  8 04:31:05 2001
    TRACE_CYCLES_PER_SEC:: 400181900
    TRACE_CPU_NUM:: 4
    TRACE_SYSNAME:: QNX
    TRACE_NODENAME:: x86quad.gp.qa
    TRACE_SYS_RELEASE:: 6.1.0
    TRACE_SYS_VERSION:: 2001/06/04-14:07:56
    TRACE_MACHINE:: x86pc
    TRACE_SYSPAGE_LEN:: 2440
-- KERNEL EVENTS --
t:0x1310da15 CPU:01 CONTROL :TIME msb:0x0000000f, lsb(offset):0x1310d81c
t:0x1310e89d CPU:01 PROCESS :PROCCREATE_NAME
    ppid:0
    pid:1
    name:./procnto-smp-instr
t:0x1310eee4 CPU:00 THREAD :THCREATE      pid:1 tid:1
t:0x1310f052 CPU:00 THREAD :THRUNNING   pid:1 tid:1
t:0x1310f144 CPU:01 THREAD :THCREATE      pid:1 tid:2
t:0x1310f201 CPU:01 THREAD :THREADY      pid:1 tid:2
t:0x1310f32f CPU:02 THREAD :THCREATE      pid:1 tid:3
t:0x1310f3ec CPU:02 THREAD :THREADY      pid:1 tid:3
t:0x1310f52d CPU:03 THREAD :THCREATE      pid:1 tid:4
t:0x1310f5ea CPU:03 THREAD :THRUNNING   pid:1 tid:4
t:0x1310f731 CPU:02 THREAD :THCREATE      pid:1 tid:5
t:0x1310f7ee CPU:02 THREAD :THRECEIVE    pid:1 tid:5
t:0x1310f921 CPU:03 THREAD :THCREATE      pid:1 tid:6
t:0x1310f9de CPU:03 THREAD :THRECEIVE    pid:1 tid:6
t:0x1310fb0b CPU:01 THREAD :THCREATE      pid:1 tid:7
t:0x1310fbc8 CPU:01 THREAD :THRECEIVE    pid:1 tid:7
t:0x1310fd1d CPU:02 THREAD :THCREATE      pid:1 tid:8
t:0x1310fdda CPU:02 THREAD :THRECEIVE    pid:1 tid:8
t:0x1310ff35 CPU:02 THREAD :THCREATE      pid:1 tid:9
t:0x1310fff2 CPU:02 THREAD :THRECEIVE    pid:1 tid:9
t:0x131100cc CPU:01 THREAD :THCREATE      pid:1 tid:10
t:0x13110189 CPU:01 THREAD :THRECEIVE    pid:1 tid:10
t:0x131102d5 CPU:03 THREAD :THCREATE      pid:1 tid:11
t:0x13110392 CPU:03 THREAD :THRECEIVE    pid:1 tid:11
t:0x1311084d CPU:01 PROCESS :PROCCREATE_NAME
    ppid:1
    pid:2
    name:proc/boot/slogger
t:0x13110c13 CPU:03 THREAD :THCREATE      pid:2 tid:1
t:0x13110ce0 CPU:03 THREAD :THRECEIVE    pid:2 tid:1

```

## **Exit status:**

-1 An error occurred.

## **Errors:**

Severe errors cause **traceprinter** to terminate; noncatastrophic errors are displayed in verbose mode (if the **-v** option is set).

## **Caveats:**

You must run **tracelogger** before running **traceprinter**. The **tracelogger** utility creates an event file containing trace data; **traceprinter** parses and prints the data in this file.

## **See also:**

**tracelogger**



*Appendix A*

---

**Kernel Call Arguments and Return Values**



The following table lists the wide- and fast-emitting mode kernel call arguments and return values for each kernel call.



Many functions listed below are internal function calls that you won't find documented in the *Library Reference*. They're included in this table because you may see them listed in your trace output. *Some* of the internal functions include:

- All functions that begin with `_`
- `InterruptDetachFunc()`
- `SignalFault()`.

All functions with a `_r` (restartable) use the same call arguments and return values as the equivalent function without the `_r`.

The `rmsg` tag indicates the contents of the message. When several are specified, the contents of `rmsg` are bytes 0-3 in the first, 4-7 in the second, and so on.

Function	Wide-emitting call arguments	Fast-emitting call arguments	Wide-emitting return values	Fast-emitting return values
<code>_bad()</code>	A1: empty, A2: empty	A1: empty, A2: empty	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<code>ChannelCreate()</code>	A1: flags, A2: empty	A1: flags, A2: empty	R1: chid, R2: empty	R1: chid, R2: empty
<code>ChannelDestroy()</code>	A1: chid, A2: empty	A1: chid, A2: empty	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<code>ClockAdjust()</code>	A1: id, A2: new->tick_count, A3: new->tick_nsec_inc	A1: id, A2: new->tick_count	R1: ret_val, R2: old->tick_count, R3: old->tick_nsec_inc	R1: ret_val, R2: old->tick_count

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>ClockId()</i>	A1: pid, A2: tid	A1: pid, A2: tid	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>ClockPeriod()</i>	A1: id, A2: new->nsec, A3: new->fract	A1: id, A2: new->nsec	R1: ret_val, R2: old->nsec, R3: old->fract	R1: ret_val, R2: old
<i>ClockTime()</i>	A1: id, A2: new(sec), A3: new(nsec)	A1: id, A2: new(sec)	R1: ret_val, R2: old(sec), R3: old(nsec)	R1: ret_val, R2: old
<i>ConnectAttach()</i>	A1: nd, A2: pid, A3: chid, A4: index, A5: flags	A1: nd, A2: pid	R1: coid, R2: empty	R1: coid, R2: empty

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>ConnectClientInfo()</i>	A1: scoid, A2: ngroups	A1: scoid, A2: ngroups	R1: ret_val, R2: info->nd, R3: info->pid, R4: info->sid, R5: flags, R6: info->ruid, R7: info->euid, R8: info->suid, R9: info->rgid, R10: info->egid, R11: info->sgid, R12: info->ngroups, R13: info->grouplist[0], R14: info->grouplist[1], R15: info->grouplist[2], R16: info->grouplist[3], R17: info->grouplist[4], R18: info->grouplist[5], R19: info->grouplist[6], R20: info->grouplist[7]	R1: ret_val, R2: info->nd
<i>ConnectDetach()</i>	A1: coid, A2: empty	A1: coid, A2: empty	R1: ret_val, R2: empty	R1: ret_val, R2: empty

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>ConnectFlags()</i>	A1: pid, A2: coid, A3: masks, A4: bits	A1: coid, A2: bits	R1: old_flags, R2: empty	R1: old_flags, R2: empty
<i>ConnectServerInfo()</i>	A1: pid, A2: coid	A1: pid, A2: coid	R1: coid, R2: info->nd, R3: info->srcnd, R4: info->pid, R5: info->tid, R6: info->chid, R7: info->scoid, R8: info->coid, R9: info->msglen, R10: info->srcmsglen, R11: info->dstmsglen, R12: info->priority, R13: info->flags, R14: info->reserved	R1: coid, R2: info->nd
<i>InterruptAttach()</i>	A1: intr, A2: handler_p, A3: area_p, A4: areabase, A5: flags	A1: intr, A2: flags	R1: int_fun_id, R2: empty	R1: int_fun_id, R2: empty
<i>InterruptDetachFunc()</i>	A1: intr, A2: handler_p	A1: intr, A2: handler_p	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>InterruptMask()</i>	A1: intr, A2: id	A1: intr, A2: id	R1: mask_level, R2: empty	R1: mask_level, R2: empty
<i>InterruptUnmask()</i>	A1: intr, A2: id	A1: intr, A2: id	R1: mask_level, R2: empty	R1: mask_level, R2: empty

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>InterruptWait()</i>	A1: flags, A2: timeout_tv_sec, A3: timeout_tv_nsec	A1: flags, A2: empty	R1: ret_val, R2: timeout_p	R1: ret_val, R2: empty
<i>MsgDeliverEvent()</i>	A1: ravid, A2: event->sigev_notify, A3: event- >sigev_notify_function_p, A4: event->sigev_value, A5: event- >sigev_notify_attributes_p	A1: ravid, A2: event->sigev_notify	R1: ret_val, R2: event_p	R1: ret_val, R2: empty
<i>MsgError()</i>	A1: ravid, A2: err	A1: ravid, A2: err	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>MsgInfo()</i>	A1: ravid, A2: info_p	A1: ravid, A2: info_p	R1: ret_val, R2: info->nd, R3: info->srcnd, R4: info->pid, R5: info->tid, R6: info->chid, R7: info->scoid, R8: info->coid, R9: info->msglen, R10: info->srcmsglen, R11: info->dstmsglen, R12: info->priority, R13: info->flags, R14: empty	R1: ret_val, R2: info->nd

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>MsgKeyData()</i>	A1: ravid, A2: op	A1: ravid, A2: op	R1: ret_val, R2: newkey	R1: ret_val, R2: newkey
<i>MsgReady()</i>	A1: ravid, A2: rmsg_p, A3: rparts, A4: offset	A1: ravid, A2: offset	R1: rbytes, R2: rmsg, R3: rmsg, R4: rmsg	R1: rbytes, R2: rmsg
<i>MsgReceivePulse()</i>	A1: chid, A2: rparts	A1: chid, A2: rparts	R1: ret_val, R2: rmsg, R3:rmsg, R4:rmsg	R1: ret_val, R2: empty
<i>MsgReceivev()</i>	A1: chid, A2: rparts	A1: chid, A2: rparts	R1: ravid, R2: rmsg, R3: rmsg, R4: rmsg, R5: info->nd, R6: info->srcnd, R7: info->pid, R8: info->tid, R9: info->chid, R10: info->scoid, R11: info->coid, R12: info->msglen, R13: info->srcmsglen, R14: info->dstmsglen, R15: info->priority, R16: info->flags, R17: empty, R18:empty	R1: ravid, R2: rmsg

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>MsgReplyv()</i>	A1: rcvid, A2: sparts, A3: status, A4: smsg, A5: smsg, A6: smsg	A1: rcvid, A2: status	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>MsgSendPulse()</i>	A1: coid, A2: priority, A3: code, A4: value	A1: coid, A2: code	R1: status, R2: empty	R1: status, R2: empty
<i>MsgSendv()</i>	A1: coid, A2: rcvid, A3: pid, A4: msg, A5: msg, A6: msg A7: sparts A8: rparts	A1: coid, A2: rcvid, A3: pid, A4: msg	R1: status, R2: rmsg, R3: rmsg, R4: rmsg	R1: status, R2:rmsg  (pid returns 0 for network processes)
<i>MsgSendvnc()</i>	A1: coid, A2: rcvid, A3: pid, A4: msg, A5: msg, A6: msg A7: sparts A8: rparts	A1: coid, A2: rcvid, A3: pid, A4: msg	R1: status, R2: rmsg, R3: rmsg, R4: rmsg	R1: status, R2:rmsg  (pid returns 0 for network processes)

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>MsgVerifyEvent()</i>	A1: ravid, A2: event->sigev_notify, A3: event- >sigev_notify_function_p, A4: event->sigev_value, A5: event- >sigev_notify_attributes_p	A1: ravid, A2: event->sigev_notify	R1: status, R2: empty	R1: status, R2: empty
<i>MsgWritev()</i>	A1: ravid, A2: sparts, A3: offset, A4: msg, A5: msg, A6: msg	A1: ravid, A2: offset	R1: wbytes, R2: empty	R1: wbytes, R2: empty

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>NetCred()</i>	A1: coid, A2: info_p	A1: coid, A2: info_p	R1: ret_val, R2: info->nd, R3: info->pid, R4: info->sid, R5: info->flags, R6: info->ruid, R7: info->euid, R8: info->suid, R9: info->rgid, R10: info->egid, R11: info->sgid, R12: info->ngroups, R13: info->grouplist[0], R14: info->grouplist[1], R15: info->grouplist[2], R16: info->grouplist[3], R17: info->grouplist[4], R18: info->grouplist[5], R19: info->grouplist[6], R20: info->grouplist[7]	R1: ret_val, R2: info->nd
<i>NetInfoscooid()</i>	A1: scooid, A2: empty	A1: scooid, A2: empty	R1: ret_val, R2: empty	R1: ret_val, R2: empty

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>NetSignalKill()</i>	A1: cred->ruid, A2: cred->euid, A3: nd, A4: pid, A5: tid, A6: signo, A7: code, A8: value	A1: pid, A2: signo	R1: empty, R2: empty	R1: status, R2: empty
<i>NetUnblock()</i>	A1: vtid, A2: empty	A1: vtid, A2: empty	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>NetVtid()</i>	A1: vtid, A2: info_p, A3: tid, A4: coid, A5: priority, A6: srcmsglen, A7: keydata, A8: srcnd, A9: dstmsglen, A10: zero	A1: vtid, A2: info_p	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>_nop()</i>	A1: dummy, A2: empty	A1: dummy, A2: empty	R1: empty, R2: empty	R1: empty, R2: empty
<i>_Ring0()</i>	A1: func_p, A2: arg_p	A1: func_p, A2: arg_p	R1: ret_val, R2: empty	R1: ret_val, R2: empty

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>SchedGet()</i>	A1: pid, A2: tid	A1: pid, A2: tid	R1: ret_val, R2: sched_priority, R3: sched_curpriority, R4: param._ss_low_priority, R5: param._ss_max_repl, R6: param._ss_repl_period.tv_sec, R7: param._ss_repl_period.tv_nsec, R8: param._ss_init_budget.tv_sec, R9: param._ss_init_budget.tv_nsec, R10: param.empty, R11: param.empty	R1: ret_val, R2: sched_priority
<i>SchedInfo()</i>	A1: pid, A2: policy	A1: pid, A2: policy	R1: ret_val, R2: priority_min, R3: priority_max, R4: interval_sec, R5: interval_nsec, R6: priority_priv,	R1: ret_val, R2: priority_max

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>SchedSet()</i>	A1: pid, A2: tid, A3: policy, A4: sched_priority, A5: sched_curpriority, A6: param._ss_low_priority, A7: param._ss_max_repl, A8: param._ss_repl_period.tv_sec, A9: param._ss_repl_period.tv_nsec, A10: param._ss_init_budget.tv_sec, A11: param._ss_init_budget.tv_nsec, A12: param.empty, A13: param.empty	A1: pid, A2: priority	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SchedYield()</i>	A1: empty, A2: empty	A1: empty, A2: empty	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SignalAction()</i>	A1: pid, A2: sigstubs_p, A3: signo, A4: act->sa_handler_p, A5: act->sa_flags, A6: act->sa_mask.bits[0], A7: act->sa_mask.bits[1]	A1: signo, A2: act->sa_handler_p	R1: ret_val, R2: oact->sa_handler_p, R3: oact->sa_flags, R4: oact->sa_mask.bits[0], R5: oact->sa_mask.bits[1]	R1: ret_val, R2: oact->sa_handler_p

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>SignalFault()</i>	A1: sigcode, A2: addr	A1: sigcode, A2: addr	R1: ret_val, R2: reg_1, R3: reg_2, R4: reg_3, R5: reg_4, R6: reg_5	R1: ret_val, R2: reg_1
<i>SignalKill()</i>	A1: nd, A2: pid, A3: tid, A4: signo, A5: code, A6: value	A1: pid, A2: signo	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SignalProcmask()</i>	A1: pid, A2: tid, A3: how, A4: sig_blocked->bits[0], A5: sig_blocked->bits[1]	A1: pid, A2: tid	R1: ret_val, R2: old_sig_blocked->bits[0], R3: old_sig_blocked->bits[1]	R1: ret_val, R2: old_sig_blocked->bits[0]
<i>SignalReturn()</i>	A1: s_p, A2: empty	A1: s_p, A2: empty	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SignalSuspend()</i>	A1: sig_blocked->bits[0], A2: sig_blocked->bits[1]	A1: sig_blocked->bits[0], A2: sig_blocked->bits[1]	R1: ret_val, R2: sig_blocked_p	R1: ret_val, R2: sig_blocked_p

*continued...*

Function	Wide-emitting call arguments	Fast-emitting call arguments	Wide-emitting return values	Fast-emitting return values
<i>SignalWaitinfo()</i>	A1: sig_wait->bits[0], A2: sig_wait->bits[1]	A1: sig_wait->bits[0], A2: sig_wait->bits[1]	R1: sig_num, R2: si_signo, R3: si_code, R4: si_errno, R5: p[0], R6: p[1], R7: p[2], R8: p[3], R9: p[4], R10: p[5], R11: p[6]	R1: sig_num, R2: si_code
<i>SyncCondvarSignal()</i>	A1: sync_p, A2: all, A3: sync->count, A4: sync->owner	A1: sync_p, A2: all	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SyncCondvarWait()</i>	A1: sync_p, A2: mutex_p, A3: sync->count, A4: sync->owner, A5: mutex->count, A6: mutex->owner	A1: sync_p, A2: mutex_p	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SyncCtl()</i>	A1: cmd, A2: sync_p, A3: data_p, A4: count, A5: owner	A1: cmd, A2: sync_p	R1: ret_val, R2: empty	R1: ret_val, R2: empty

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>SyncDestroy()</i>	A1: sync_p, A2: count, A3: owner	A1: sync_p, A2: owner	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SyncMutexLock()</i>	A1: sync_p, A2: count, A3: owner	A1: sync_p, A2: owner	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SyncMutexRevive()</i>	A1: sync_p, A2: count, A3: owner	A1: sync_p, A2: owner	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SyncMutexUnlock()</i>	A1: sync_p, A2: count, A3: owner	A1: sync_p, A2: owner	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SyncSemPost()</i>	A1: sync_p, A2: count, A3: owner	A1: sync_p, A2: count	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SyncSemWait()</i>	A1: sync_p, A2: try, A3: count, A4: owner	A1: sync_p, A2: count	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SyncTypeCreate()</i>	A1: type, A2: sync_p, A3: count, A4: owner, A5: protocol, A6: flags, A7: prioceiling, A8: clockid	A1: type, A2: sync_p	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>SysCpupageGet()</i>	A1: index, A2: empty	A1: index, A2: empty	R1: ret_val, R2: empty	R1: ret_val, R2: empty

*continued...*

---

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>SysCpupageSet()</i>	A1: index, A2: value	A1: index, A2: value	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>ThreadDestroyAll()</i>	A1: empty, A2: empty	A1: empty, A2: empty	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>ThreadCancel()</i>	A1: tid, A2: canstub_p	A1: tid, A2: canstub_p	R1: ret_val, R2: empty	R1: ret_val, R2: empty

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>ThreadCreate()</i>	A1: pid, A2: func_p, A3: arg_p, A4: flags, A5: stacksize, A6: stackaddr_p, A7: exitfunc_p, A8: policy, A9: sched_priority, A10: sched_curpriority, A11: param...ss_low_priority, A12: param...ss_max_repl, A13: param...ss_repl_period.tv_sec, A14: param...ss_repl_period.tv_nsec, A15: param...ss_init_budget.tv_sec, A16: param...ss_init_budget.tv_nsec, A17: param.empty, A18: param.empty, A19: guardsize, A20: empty, A21: empty, A22: empty	A1: func_p, A2: arg_p	R1: thread_id, R2: owner	R1: thread_id, R2: owner
<i>ThreadCtl()</i>	A1: cmd, A2: data_p	A1: cmd, A2: data_p	R1: ret_val, R2: empty	R1: ret_val, R2: empty

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>ThreadDestroy()</i>	A1: tid, A2: priority, A3: status_p	A1: tid, A2: status_p	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>ThreadDetach()</i>	A1: tid, A2: empty	A1: tid, A2: empty	R1: ret_val, R2: empty	R1: ret_val, R2: empty
<i>ThreadJoin()</i>	A1: tid, A2: status_pp	A1: tid, A2: status_pp	R1: ret_val, R2: status_p	R1: ret_val, R2: status_p
<i>TimerAlarm()</i>	A1: id, A2: itime->nsec(sec), A3: itime->nsec(nsec), A4: itime- >interval_nsec(sec), A5: itime- >interval_nsec(nsec)	A1: id, A2: itime->nsec(sec)	R1: ret_val, R2: oitime->nsec(sec), R3: oitime->nsec(nsec), R4: oitime- >interval_nsec(sec), R5: oitime- >interval_nsec(nsec)	R1: ret_val, R2: oitime->nsec(sec)
<i>TimerCreate()</i>	A1: id, A2: event->sigev_notify, A3: event- >sigev_notify_function_p, A4: event->sigev_value, A5: event- >sigev_notify_attributes_p	A1: id, A2: event->sigev_notify	R1: timer_id, R2: empty	R1: timer_id, R2: empty
<i>TimerDestroy()</i>	A1: id, A2: empty	A1: id, A2: empty	R1: ret_val, R2: empty	R1: ret_val, R2: empty

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>TimerInfo()</i>	A1: pid, A2: id, A3: flags, A4: info_p	A1: pid, A2: id	R1: prev_id, R2: info->itime.nsec(sec), R3: info- >itime.nsec(nsec), R4: info- >itime.interval_nsec(sec), R5: info- >itime.interval_nsec(nsec), R6: info->otime.nsec(sec), R7: info- >otime.nsec(nsec), R8: info- >otime.interval_nsec(sec), R9: info- >otime.interval_nsec(nsec), R10: info->flags, R11: info->tid, R12: info->notify, R13: info->clockid, R14: info->overruns, R15: info- >event.sigev_notify, R16: info- >event.sigev_notify_function_p, R17: info- >event.sigev_value, R18: info- >event.sigev_notify_attributes_p	R1: prev_id, R2: info->itime.nsec(sec)

*continued...*

<b>Function</b>	<b>Wide-emitting call arguments</b>	<b>Fast-emitting call arguments</b>	<b>Wide-emitting return values</b>	<b>Fast-emitting return values</b>
<i>TimerSettime()</i>	A1: id, A2: flags, A3: itime->nsec(sec), A4: itime->nsec(nsec), A5: itime- >interval_nsec(sec), A6: itime- >interval_nsec(nsec)	A1: id, A2: itime->nsec(sec)	R1: ret_val, R2: oitime->nsec(sec), R3: oitime->nsec(nsec), R4: oitime- >interval_nsec(sec), R5: oitime- >interval_nsec(nsec)	R1: ret_val, R2: oitime->nsec(sec)
<i>TimerTimeout()</i>	A1: id, A2: timeout_flags, A3: ntime(sec), A4: ntime(nsec), A5: event->sigev_notify, A6: event- >sigev_notify_function_p, A7: event->sigev_value, A8: event- >sigev_notify_attributes_p	A1: timeout_flags, A2: ntime(sec)	R1: prev_timeout_flags, R2: otime(sec), R3: otime(nsec)	R1: prev_timeout_flags, R2: otime(sec)
<i>TraceEvent()</i>	A1: mode, A2: class[header], A3: event[time_off], A4: data_1, A5: data_2	A1: mode, A2: class	R1: ret_val, R2: empty	R1: ret_val, R2: empty

## ***Index***

---

### **!**

`_intrspin_t` 88

### **B**

buffer 38  
controlling 90

### **C**

circular linked list 21  
combine events 37, 38  
configuring 30

### **D**

daemon mode 29  
data capture 5, 27, 31  
data interpretation 5, 36–39  
data reduction 43

dynamic rules filter 43  
controlling 90

### **E**

events 15, 37, 38  
capturing 29  
interpreting 37  
user-generated 90

### **F**

fast mode 30, 43  
controlling 90  
filters 43  
controlling 90  
functions 87

### **I**

I/O privileges

- requesting 88
- Instrumented Kernel 5, 21, 30, 38
- Instrumented kernel
  - trace support
    - (**traceprinter**) 129
- interlacing 38
- InterruptHookTrace()* **88**

## K

- Kernel Buffer 5
- kernel buffer 21, 22

## L

- learning 51
- library 36
- log 30

## N

- Neutrino kernel functions 87
  - TraceEvent()* 98
- normal mode 29
- \_NTO\_INTR\_FLAGS\_END 89
- \_NTO\_TCTL\_IO 88
- \_NTO\_TRACE\_\* 90

## P

- post-processing 43

## S

- SAT 3, 5
- simple events 37
- starting 30, 31
- static rules filter 43
  - controlling 90
- structures 37

## T

- ThreadCtl()* 88
- threads 15
- trace\*()* 87
- TraceEvent()* **90, 98**
- tracelog 30
- tracelogger** 27–31
- traceparser()* **107**
- traceparser\_cs()* **109**
- TRACEPARSER.INFO\_\* 116
- traceprinter** 129
- Tracing support
  - trace results
    - displaying (**traceprinter**) 129
- troubleshooting 83
- tutorial 51

## U

- utilities 125

## **W**

wide mode 30, 43  
controlling 90