System Analysis Toolkit (SAT) User's Guide

QNX Software Systems Limited
1001 Farrar Road
Ottawa, Ontario
K2K 0B3
Canada

Voice: +1 613 591-0931
Fax: +1 613 591-3579
Email: info@qnx.com
Web: http://www.qnx.com/

Electronic edition published: March 06, 2020

Contents

About This Guide

The QNX Neutrino System Analysis Toolkit *User's Guide* describes how to use the instrumented microkernel to obtain a detailed analysis of what's happening in an entire QNX Neutrino system. This guide contains the following sections and chapters:

To find out about:	Go to:
What the SAT is, what it can do for you, and how it works	*Introduction*
What generates events	*Events and the Kernel*
How the kernel buffers data	*Kernel Buffer Management*
How to save data	*Capturing Trace Data*
Different ways to reduce the amount of data	*Filtering*
What the data tells you	*Interpreting Trace Data*
Examples of filtering the trace data	*Tutorials*
Sample data-capture and parsing programs	*Sample Programs*
What specific events return	*Current Trace Events and Data*

Typographical conventions

Throughout this manual, we use certain typographical conventions to distinguish technical terms. In general, the conventions we use conform to those found in IEEE POSIX publications.

The following table summarizes our conventions:

Reference	Example
Code examples	`if( stream == NULL)`
Command options	`-lR`
Commands	`make`
Constants	NULL
Data types	`unsigned short`
Environment variables	*PATH*
File and pathnames	**/dev/null**
Function names	*exit()*
Keyboard chords	**Ctrl–Alt–Delete**
Keyboard input	`Username`
Keyboard keys	**Enter**
Program output	`login:`
Variable names	*stdin*
Parameters	*parm1*
User-interface components	**Navigator**
Window title	**Options**

We use an arrow in directions for accessing menu items, like this:

You'll find the Other... menu item under **Perspective** → **Show View**.

We use notes, cautions, and warnings to highlight important messages:

Notes point out something important or useful.

CAUTION: Cautions tell you about commands or procedures that may have unwanted or undesirable side effects.

DANGER: Warnings tell you about commands or procedures that could be dangerous to your files, your hardware, or even yourself.

Note to Windows users

In our documentation, we typically use a forward slash (/) as a delimiter in pathnames, including those pointing to Windows files. We also generally follow POSIX/UNIX filesystem conventions.

Technical support

Technical assistance is available for all supported products.

To obtain technical support for any QNX product, visit the Support area on our website (*www.qnx.com*). You'll find a wide range of support options, including community forums.

Chapter 1
Introduction

In many 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:

- ease of use
- insight into system activity
- high performance and efficiency with low overhead

What can the SAT do for you?

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 results 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 QNX 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.

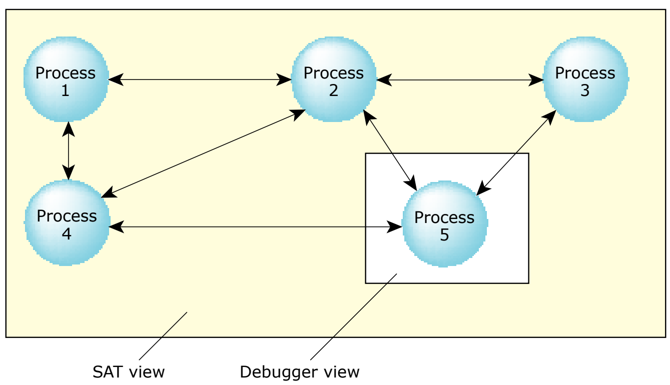

Figure 1: The SAT view and the debugger view.

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.

Components of the SAT

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

- *instrumented kernel*
- *kernel buffer management*
- *data-capture program (`tracelogger`)*
- *data interpretation (e.g., `traceprinter`)*

You can also trace and analyze events under control of the *Integrated Development Environment*.

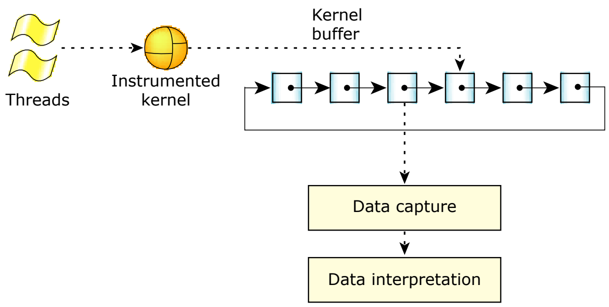

Figure 2: Overall view of the SAT.

Instrumented kernel

The QNX Neutrino instrumented kernel includes a small, highly efficient event-gathering module. 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 ring 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 trace 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 trace buffers are organized as a ring. As the buffer data reaches a high-water mark (about 70% full in linear mode, or 90% in ring mode), the instrumented kernel module raises an _NTO_HOOK_TRACE synthetic interrupt to notify the data-capture program, passing the index of the buffer. The data-capture program can then retrieve the buffer and save it in 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's once again available for use by the kernel.

Data-capture program (`tracelogger`)

The QNX Neutrino RTOS includes a `tracelogger` that you can use to capture data. This service receives events from the instrumented kernel and saves them in a file or sends them to a device for later analysis.

The data-capture utilities require the PROCMGR_AID_TRACE ability enabled in order to allocate buffer memory. For more information about abilities, see the entry for *procmgr_ability()* in the QNX Neutrino *C Library Reference*.

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

You can also control tracing from your application (e.g., to turn tracing on just for a problematic area) with the *TraceEvent()* kernel call. This function has over 30 different commands that let you:

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

For more information, see the *Capturing Trace Data* chapter in this guide, the entry for `tracelogger` in the *Utilities Reference*, and the entry for *TraceEvent()* in the QNX Neutrino *C Library Reference*.

Data interpretation (e.g., `traceprinter`)

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're emitted by the kernel. This utility uses the **libtraceparser** library. You can also use the API to create an interface to do the following offline or in real time:

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

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 letting you 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

For more information, see the *Interpreting Trace Data* chapter in this guide, as well as the entry for `traceprinter` in the *Utilities Reference*.

Integrated Development Environment

The QNX Momentics Tool Suite's IDE provides a graphical interface that you can use to capture and examine tracing events. The IDE lets you filter events, zoom in on ranges of them, examine the associated data, save subsets of events, and more.

For more information, see the “Analyzing System Behavior” chapter of the IDE *User's Guide*.

Chapter 2
Events and the Kernel

The QNX Neutrino 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

Processes that are running on QNX Neutrino can 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:

Thread	Priority
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.
t9	...	...	...

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.

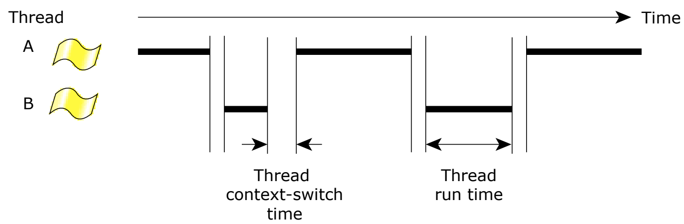

Figure 3: Thread context switching.

In some cases, two or more threads 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.”

Restarting threads

In order to achieve maximum responsiveness, much of the QNX Neutrino microkernel is fully preemptible. 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.

Simple and combine events

Most events can be described in a single event buffer slot; we call these *simple events*. When there's too much information to describe the event in a single buffer slot, the event is described in multiple event buffer slots; we call this a *combine event*. The event 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 *Interpreting Trace Data* chapter.

Fast and wide modes

You can gather data for events in the following modes:

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. Most of the time, it doesn't exceed *four* 16-byte spaces.

Fast mode

The instrumented kernel uses *only one* buffer slot per event.

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

Fast mode doesn't simply clip the tail end of the event data that you'd get in wide mode; fast mode summarizes the most important aspects of the event in a single buffer slot. Thus, the first element of an event in wide mode might not be the same as the same event in fast mode.

You can set fast and wide mode for all classes, specific classes, and even specific events in a class; some can be fast while others are wide. We'll describe how to set this in the *Capturing Trace Data* chapter.

For the specific output differences between fast and wide mode, see the *Current Trace Events and Data* appendix.

Classes and events

There can be a lot of events in even a small trace, so they're organized into *classes* to make them easier for you to manage:

- *Communication class:* _NTO_TRACE_COMM
- *Control class:* _NTO_TRACE_CONTROL
- *Interrupt classes:* _NTO_TRACE_INTENTER, _NTO_TRACE_INTEXIT, _NTO_TRACE_INT_HANDLER_ENTER, *and* _NTO_TRACE_INT_HANDLER_EXIT
- *Kernel-call classes:* _NTO_TRACE_KERCALLENTER, _NTO_TRACE_KERCALLEXIT, *and* _NTO_TRACE_KERCALLINT
- *Process class:* _NTO_TRACE_PROCESS
- QNX Unified Instrumentation Platform class: _NTO_TRACE_QUIP
- *Security class:* _NTO_TRACE_SEC
- *System class:* _NTO_TRACE_SYSTEM
- *Thread class:* _NTO_TRACE_THREAD
- *User class:* _NTO_TRACE_USER
- *Virtual thread class:* _NTO_TRACE_VTHREAD

(The **<sys/trace.h>** header file also defines an _NTO_TRACE_EMPTY class, but it's a placeholder and isn't currently used.)

The sections that follow list the events for each class, along with a description of when the events are emitted, as well as the labels that `traceprinter` and the IDE use to identify the events.

For information about the data for each event, see the *Current Trace Events and Data* appendix.

Communication class: _NTO_TRACE_COMM

The _NTO_TRACE_COMM class includes events related to communication.

Event	`traceprinter` label	IDE label	Emitted when:
_NTO_TRACE_COMM_ERROR	`MSG_ERROR`	`Error`	A client is unblocked because of a call to *MsgError()*
_NTO_TRACE_COMM_REPLY	`REPLY_MESSAGE`	`Reply`	A reply is sent
_NTO_TRACE_COMM_RMSG	`REC_MESSAGE`	`Receive Message`	A message is received
_NTO_TRACE_COMM_RPULSE	`REC_PULSE`	`Receive Pulse`	A pulse is received
_NTO_TRACE_COMM_SIGNAL	`SIGNAL`	`Signal`	A signal is received
_NTO_TRACE_COMM_SMSG	`SND_MESSAGE`	`Send Message`	A message is sent

Event	`traceprinter` label	IDE label	Emitted when:
_NTO_TRACE_COMM_SPULSE	`SND_PULSE`	`Send Pulse`	A pulse is sent
_NTO_TRACE_COMM_SPULSE_DEA	`SND_PULSE_DEA`	`Death Pulse`	A _PULSE_CODE_COIDDEATH pulse is sent
_NTO_TRACE_COMM_SPULSE_DIS	`SND_PULSE_DIS`	`Disconnect Pulse`	A _PULSE_CODE_DISCONNECT pulse is sent
_NTO_TRACE_COMM_SPULSE_EXE	`SND_PULSE_EXE`	`Sigevent Pulse`	A SIGEV_PULSE is sent
_NTO_TRACE_COMM_SPULSE_QUN	`SND_PULSE_QUN`	`QNet Unblock Pulse`	A _PULSE_CODE_NET_UNBLOCK pulse is sent
_NTO_TRACE_COMM_SPULSE_UN	`SND_PULSE_UN`	`Unblock Pulse`	A _PULSE_CODE_UNBLOCK pulse is sent

Control class: _NTO_TRACE_CONTROL

The _NTO_TRACE_CONTROL class includes events related to the control of tracing itself.

Event	`traceprinter` label	IDE label	Emitted when:
_NTO_TRACE_CONTROLBUFFER	`BUFFER`	`Buffer`	The instrumented kernel starts filling a new buffer
_NTO_TRACE_CONTROLTIME	`TIME`	`Time`	The 32 Least Significant Bits (LSB) part of the 64-bit clock rolls over, or the kernel emits an _NTO_TRACE_CONTROLBUFFER event

The purpose of emitting _NTO_TRACE_CONTROLBUFFER events is to help `tracelogger` and the IDE track the buffers and determine if any buffers have been dropped. The instrumented kernel emits an _NTO_TRACE_CONTROLTIME event at the same time to keep the IDE in sync (in case a dropped buffer contained an _NTO_TRACE_CONTROLTIME event for a rollover of the clock).

Interrupt classes: _NTO_TRACE_INTENTER, _NTO_TRACE_INTEXIT, _NTO_TRACE_INT_HANDLER_ENTER, and _NTO_TRACE_INT_HANDLER_EXIT

These classes track interrupts.

Class	traceprinter label	IDE label	Emitted when:
_NTO_TRACE_INTENTER	INT_ENTR	Entry	Overall processing of an interrupt begins
_NTO_TRACE_INTEXIT	INT_EXIT	Exit	Overall processing of an interrupt ends
_NTO_TRACE_INT_HANDLER_ENTER	INT_HANDLER_ENTR	Handler Entry	Entering an interrupt handler
_NTO_TRACE_INT_HANDLER_EXIT	INT_HANDLER_EXIT	Handler Exit	Exiting an interrupt handler

The expected sequence is:

```
INTR_ENTER
    INTR_HANDLER_ENTER
    INTR_HANDLER_EXIT
    INTR_HANDLER_ENTER
    INTR_HANDLER_EXIT
INT_EXIT
```

_NTO_TRACE_INT is a pseudo-class that comprises all of the interrupt classes.

The “event” is an interrupt vector number, in the range from _NTO_TRACE_INTFIRST through _NTO_TRACE_INTLAST.

Kernel-call classes: _NTO_TRACE_KERCALLENTER, _NTO_TRACE_KERCALLEXIT, and _NTO_TRACE_KERCALLINT

These classes track kernel calls.

- _NTO_TRACE_KERCALLENTER and _NTO_TRACE_KERCALLEXIT track the entrances to and exits from kernel calls.
- _NTO_TRACE_KERCALLINT tracks interrupted kernel calls. When we exit the kernel, we check to see if the kernel call arguments are valid. If so, then we log an _NTO_TRACE_KERCALLEXIT event with the parameters. If not, then we log an _NTO_TRACE_KERCALLINT event with no parameters. If you get an EINTR return code from your kernel call, you'll also see an _NTO_TRACE_KERCALLINT event in the trace log.

_NTO_TRACE_KERCALL is a pseudo-class that comprises all these classes.

The `traceprinter` labels for these classes are `KER_CALL`, `KER_EXIT`, and `INT_CALL`, followed by an uppercase version of the kernel call; the IDE labels consist of the kernel call, followed by `Enter`, `Exit`, or `INT`.

The event type is ORed with _NTO_TRACE_KERCALL64 if the event information includes 64-bit data types. For the most part this happens when you're tracing a 64-bit process, but it's possible for a 32-bit process to call *MsgDeliverEvent()* with the 64-bit form of a `struct sigevent` (e.g., when a 64-bit client interacts with a 32-bit server). When you're doing a wide-mode trace, the kernel dumps the contents of the `struct sigevent`, so _NTO_TRACE_KERCALL64 would be ORed into the event type.

Most of the events in these classes correspond in a fairly obvious way to the kernel calls; some correspond to internal functions:

Event	Kernel call
__KER_BAD	—
__KER_CACHE_FLUSH (QNX Neutrino 6.6 or later)	*CacheFlush()*
__KER_CHANCON_ATTR	*ChannelConnectAttr()*
__KER_CHANNEL_CREATE	*ChannelCreate()*
__KER_CHANNEL_DESTROY	*ChannelDestroy()*
__KER_CLOCK_ADJUST	*ClockAdjust()*
__KER_CLOCK_ID	*ClockId()*
__KER_CLOCK_PERIOD	*ClockPeriod()*
__KER_CLOCK_TIME	*ClockTime()*
__KER_CONNECT_ATTACH	*ConnectAttach()*
__KER_CONNECT_CLIENT_INFO	*ConnectClientInfo()*
__KER_CONNECT_DETACH	*ConnectDetach()*
__KER_CONNECT_FLAGS	*ConnectFlags()*
__KER_CONNECT_SERVER_INFO	*ConnectServerInfo()*
__KER_INTERRUPT_ATTACH	*InterruptAttach()*
__KER_INTERRUPT_CHARACTERISTIC (QNX Neutrino 6.6 or later)	*InterruptCharacteristic()*
__KER_INTERRUPT_DETACH	*InterruptDetach()*
__KER_INTERRUPT_DETACH_FUNC	—
__KER_INTERRUPT_MASK	*InterruptMask()*

Event	Kernel call
__KER_INTERRUPT_UNMASK	*InterruptUnmask()*
__KER_INTERRUPT_WAIT	*InterruptWait()*
__KER_MSG_CURRENT	*MsgCurrent()*
__KER_MSG_DELIVER_EVENT	*MsgDeliverEvent()*
__KER_MSG_ERROR	*MsgError()*
__KER_MSG_INFO	*MsgInfo()*
__KER_MSG_KEYDATA	*MsgKeyData()*
__KER_MSG_PAUSE (QNX Neutrino 6.6 or later)	*MsgPause()*
__KER_MSG_READV	*MsgRead()*, *MsgReadv()*
__KER_MSG_RECEIVEPULSEV	*MsgReceivePulse()*, *MsgReceivePulsev()*
__KER_MSG_RECEIVEV	*MsgReceive()*, *MsgReceivev()*
__KER_MSG_REPLYV	*MsgReply()*, *MsgReplyv()*
__KER_MSG_SENDV	*MsgSend()*, *MsgSendv()*, and *MsgSendvs()*
__KER_MSG_SENDVNC	*MsgSendnc()*, *MsgSendvnc()*, and *MsgSendvsnc()*
__KER_MSG_SEND_PULSE	*MsgSendPulse()*
__KER_MSG_SEND_PULSEPTR	*MsgSendPulsePtr()*
__KER_MSG_VERIFY_EVENT	*MsgVerifyEvent()*
__KER_MSG_WRITEV	*MsgWrite()*, *MsgWritev()*
__KER_NET_CRED	*NetCred()*
__KER_NET_INFOSCOID	*NetInfoScoid()*
__KER_NET_SIGNAL_KILL	*NetSignalKill()*
__KER_NET_UNBLOCK	*NetUnblock()*
__KER_NET_VTID	*NetVtid()*
__KER_NOP	None; forces a thread into the kernel so that scheduling can take place
__KER_RING0 (not generated in QNX Neutrino 6.3.0 or later)	*__Ring0()*
__KER_SCHED_CTL	*SchedCtl()*

Event	Kernel call
__KER_SCHED_GET	*SchedGet()*
__KER_SCHED_INFO	*SchedInfo()*
__KER_SCHED_SET	*SchedSet()*
__KER_SCHED_WAYPOINT	*SchedWaypoint()*
__KER_SCHED_YIELD	*SchedYield()*
__KER_SIGNAL_ACTION	*SignalAction()*
__KER_SIGNAL_FAULT	—
__KER_SIGNAL_KILL	*SignalKill()*
__KER_SIGNAL_KILL_SIGVAL	*SignalKillSigval()*
__KER_SIGNAL_PROCMASK	*SignalProcmask()*
__KER_SIGNAL_RETURN	*SignalReturn()*
__KER_SIGNAL_SUSPEND	*SignalSuspend()*
__KER_SIGNAL_WAITINFO	*SignalWaitInfo()*
__KER_SYNC_CONDVAR_SIGNAL	*SyncCondvarSignal()*
__KER_SYNC_CONDVAR_WAIT	*SyncCondvarWait()*
__KER_SYNC_CREATE	*SyncCreate()*, *SyncTypeCreate()*
__KER_SYNC_CTL	*SyncCtl()*
__KER_SYNC_DESTROY	*SyncDestroy()*
__KER_SYNC_MUTEX_LOCK	*SyncMutexLock()*
__KER_SYNC_MUTEX_REVIVE	*SyncMutexRevive()*
__KER_SYNC_MUTEX_UNLOCK	*SyncMutexUnlock()*
__KER_SYNC_SEM_POST	*SyncSemPost()*
__KER_SYNC_SEM_WAIT	*SyncSemWait()*
__KER_SYS_CPUPAGE_GET	—
__KER_SYS_CPUPAGE_SET	—
__KER_SYS_SRANDOM	*SysSrandom()*

Event	Kernel call
__KER_THREAD_CANCEL	*ThreadCancel()*
__KER_THREAD_CREATE	*ThreadCreate()*
__KER_THREAD_CTL	*ThreadCtl()*
__KER_THREAD_DESTROY	*ThreadDestroy()*
__KER_THREAD_DESTROYALL	—
__KER_THREAD_DETACH	*ThreadDetach()*
__KER_THREAD_JOIN	*ThreadJoin()*
__KER_TIMER_ALARM	*TimerAlarm()*
__KER_TIMER_CREATE	*TimerCreate()*
__KER_TIMER_DESTROY	*TimerDestroy()*
__KER_TIMER_INFO	*TimerInfo()*
__KER_TIMER_SETTIME	*TimerSettime()*
__KER_TIMER_TIMEOUT	*TimerTimeout()*
__KER_TRACE_EVENT	*TraceEvent()*

Process class: _NTO_TRACE_PROCESS

The _NTO_TRACE_PROCESS class includes events related to the creation and destruction of processes.

Event	**traceprinter** label	IDE label	Emitted when:
_NTO_TRACE_PROCCREATE	`PROCCREATE`	`Create Process`	A process is created
_NTO_TRACE_PROCCREATE_NAME	`PROCCREATE_NAME`	`Create Process Name`	A newly created process is given a name.
_NTO_TRACE_PROCDESTROY	`PROCDESTROY`	`Destroy Process`	A process is destroyed
_NTO_TRACE_PROCDESTROY_NAME	—	—	(Not currently used)
_NTO_TRACE_PROCTHREAD_NAME	`PROCTHREAD_NAME`	`Thread Name`	A name is assigned to a thread

Security class: _NTO_TRACE_SEC

(QNX Neutrino 7.0 or later) The _NTO_TRACE_SEC class includes events related to security.

Event	**traceprinter** label	IDE label	Emitted when:
_NTO_TRACE_SEC_ABLE	`ABLE`	`Able`	A process is tested for having a process manager ability; see *procmgr_ability()* in the QNX Neutrino *C Library Reference*.
_NTO_TRACE_SEC_ABLE_LOOKUP	`ABLE_LOOKUP`	`Able_lookup`	A dynamic process manager ability is being created or looked up; see *procmgr_ability_create()* and *procmgr_ability_lookup()* in the QNX Neutrino *C Library Reference*.
_NTO_TRACE_SEC_PATH_ATTACH	`PATH_ATTACH`	`Path_attach`	When a link is made or fails to be made to the path space via *resmgr_attach()* or *pathmgr_link()*
_NTO_TRACE_SEC_QNET_CONNECT	`QNET_CONNECT`	`Qnet_connect`	When a remote process attempts to connect to a channel over Qnet and a security policy is in effect. Note that `procnto` doesn't emit these events; they're part of mandatory access control (MAC).

For more information, see the *Security Developer's Guide*.

System class: _NTO_TRACE_SYSTEM

The _NTO_TRACE_SYSTEM class includes events related to the system as a whole.

Event	**traceprinter** label	IDE label	Emitted when:
_NTO_TRACE_SYS_ADDRESS	`ADDRESS`	`Address`	A breakpoint is hit
_NTO_TRACE_SYS_APS_BNKR	`APS_BANKRUPTCY`	`APS Bankruptcy`	An adaptive partition exceeded its critical budget
_NTO_TRACE_SYS_APS_BUDGETS	`APS_NEW_BUDGET`	`APS Budgets`	*SchedCtl()* is called with a command of SCHED_APS_CREATE_PARTITION or SCHED_APS_MODIFY_PARTITION. Also emitted automatically when the adaptive partitioning scheduler clears a critical budget as part of handling a bankruptcy.

Event	`traceprinter` label	IDE label	Emitted when:
_NTO_TRACE_SYS_APS_NAME	`APS_NAME`	`APS Name`	*SchedCtl()* is called with a command of SCHED_APS_CREATE_PARTITION
_NTO_TRACE_SYS_FUNC_ENTER	`FUNC_ENTER`	`Function Enter`	A function that's instrumented for profiling is entered
_NTO_TRACE_SYS_FUNC_EXIT	`FUNC_EXIT`	`Function Exit`	A function that's instrumented for profiling is exited
_NTO_TRACE_SYS_IPI, _NTO_TRACE_SYS_IPI_64	`IPI`	`IPI`	An interprocessor interrupt is received
_NTO_TRACE_SYS_MAPNAME, _NTO_TRACE_SYS_MAPNAME_64	`MAPNAME`	`Map Name`	*dlopen()* is called. Note that the kernel generates only _NTO_TRACE_SYS_MAPNAME_64 events, even for 32-bit processes.
_NTO_TRACE_SYS_MMAP	`MMAP`	`MMap`	*mmap()* or *mmap64()* is called
_NTO_TRACE_SYS_MUNMAP	`MUNMAP`	`MMUnmap`	*munmap()* is called
_NTO_TRACE_SYS_PAGEWAIT	`PAGEWAIT`	`Pagewait`	A page fault is being handled
_NTO_TRACE_SYS_PATHMGR	`PATHMGR_OPEN`	`Path Manager`	An operation involving a path name—such as *open()*—that's routed via the **libc** connect function occurs. The connect function sends a message to `procnto` to resolve the path and find the set of resource managers that could potentially match the path. It's upon receiving this message that `procnto` emits this event.
_NTO_TRACE_SYS_POWER	`POWER`	`Power`	A CPU enters or exits idle mode
_NTO_TRACE_SYS_PROFILE, _NTO_TRACE_SYS_PROFILE_64	`PROFILE`	`Profile`	Every clock tick, if statistical profiling is enabled
_NTO_TRACE_SYS_RUNSTATE	`RUNSTATE`	`Runstate`	The runstate for a CPU changes
_NTO_TRACE_SYS_SLOG	`SLOG`	`System Log`	A message is written to the system log
_NTO_TRACE_SYS_TIMER	`TIMER`	`Timer`	A timer expires

You can use the following convenience functions to insert certain System events into the trace data:

trace_func_enter()

Insert an _NTO_TRACE_SYS_FUNC_ENTER event for a function

trace_func_exit()

Insert an _NTO_TRACE_SYS_FUNC_EXIT event for a function

trace_here()

Insert an _NTO_TRACE_SYS_ADDRESS event for the current address

Thread class: _NTO_TRACE_THREAD

The _NTO_TRACE_THREAD class includes events related to state changes for threads.

Event	`traceprinter` label	IDE label	Emitted when a thread:
_NTO_TRACE_THCONDVAR	`THCONDVAR`	`Condvar`	Enters the CONDVAR state
_NTO_TRACE_THCREATE	`THCREATE`	`Create Thread`	Is created
_NTO_TRACE_THDEAD	`THDEAD`	`Dead`	Enters the DEAD state
_NTO_TRACE_THDESTROY	`THDESTROY`	`Destroy Thread`	Is destroyed
_NTO_TRACE_THINTR	`THINTR`	`Interrupt`	Enters the INTERRUPT state
_NTO_TRACE_THJOIN	`THJOIN`	`Join`	Enters the JOIN state
_NTO_TRACE_THMUTEX	`THMUTEX`	`Mutex`	Enters the MUTEX state
_NTO_TRACE_THNANOSLEEP	`THNANOSLEEP`	`NanoSleep`	Enters the NANOSLEEP state
_NTO_TRACE_THNET_REPLY	`THNET_REPLY`	`NetReply`	Enters the NET_REPLY state
_NTO_TRACE_THNET_SEND	`THNET_SEND`	`NetSend`	Enters the NET_SEND state
_NTO_TRACE_THREADY	`THREADY`	`Ready`	Enters the READY state
_NTO_TRACE_THRECEIVE	`THRECEIVE`	`Receive`	Enters the RECEIVE state
_NTO_TRACE_THREPLY	`THREPLY`	`Reply`	Enters the REPLY state
_NTO_TRACE_THRUNNING	`THRUNNING`	`Running`	Enters the RUNNING state
_NTO_TRACE_THSEM	`THSEM`	`Semaphore`	Enters the SEM state
_NTO_TRACE_THSEND	`THSEND`	`Send`	Enters the SEND state
_NTO_TRACE_THSIGSUSPEND	`THSIGSUSPEND`	`SigSuspend`	Enters the SIGSUSPEND state

Event	`traceprinter` label	IDE label	Emitted when a thread:
_NTO_TRACE_THSIGWAITINFO	`THSIGWAITINFO`	`SigWaitInfo`	Enters the SIGWAITINFO state
_NTO_TRACE_THSTACK	`THSTACK`	`Stack`	Enters the STACK state
_NTO_TRACE_THSTOPPED	`THSTOPPED`	`Stopped`	Enters the STOPPED state
_NTO_TRACE_THWAITCTX	`THWAITCTX`	`WaitCtx`	Enters the WAITCTX state
_NTO_TRACE_THWAITPAGE	`THWAITPAGE`	`WaitPage`	Enters the WAITPAGE state
_NTO_TRACE_THWAITTHREAD	`THWAITTHREAD`	`WaitThread`	Enters the WAITTHREAD state

If your system includes the adaptive partitioning scheduler module, the data for these events includes the partition ID and scheduling flags. For more information about adaptive partitioning, see the Adaptive Partitioning *User's Guide*.

For more information about thread states, see “Thread life cycle” in the QNX Neutrino Microkernel chapter of the *System Architecture* guide.

User class: _NTO_TRACE_USER

The _NTO_TRACE_USER class includes custom events that your program creates.

You can create these events by calling one of the following convenience functions:

trace_logb()

Insert a user combine trace event

trace_logf()

Insert a user string trace event

trace_logi()

Insert a user simple trace event

trace_nlogf()

Insert a user string trace event, specifying a maximum string length

trace_vnlogf()

Insert a user string trace event, using a variable argument list

or by calling *TraceEvent()* directly, with one of the following commands:

- _NTO_TRACE_INSERTSUSEREVENT to create a simple event containing a small amount of data
- _NTO_TRACE_INSERTCUSEREVENT to create a combine event containing an arbitrary amount of data

The *len* argument for the _NTO_TRACE_INSERTCUSEREVENT command is the number of *integers* (not bytes) in the passed buffer.

- _NTO_TRACE_INSERTUSRSTREVENT to create an event containing a null-terminated string

The event must be in the range from _NTO_TRACE_USERFIRST through _NTO_TRACE_USERLAST, but you can decide what each event means.

The `traceprinter` label for these events is `USREVENT`; the IDE label is `User Event`. In both cases, this label is followed by the event type, expressed as an integer.

Virtual thread class: _NTO_TRACE_VTHREAD

The _NTO_TRACE_VTHREAD class includes events related to state changes for *virtual threads*, special objects related to Transparent Distributed Processing (TDP) over Qnet.

The kernel often keeps pointers from different data structures to relevant threads. When those threads are off-node via Qnet, there isn't a local thread object to represent them, so the kernel creates a virtual thread object.

The events for virtual threads are similar to those for normal threads, but virtual threads don't go through the same set of state transitions that normal threads do:

Event	`traceprinter` label	IDE label	Emitted when a virtual thread:
_NTO_TRACE_VTHCONDVAR	`VTHCONDVAR`	`VCondvar`	Enters the CONDVAR state
_NTO_TRACE_VTHCREATE	`VTHCREATE`	`Create VThread`	Is created
_NTO_TRACE_VTHDEAD	`VTHDEAD`	`VDead`	Enters the DEAD state
_NTO_TRACE_VTHDESTROY	`VTHDESTROY`	`Destroy VThread`	Is destroyed
_NTO_TRACE_VTHINTR	`VTHINTR`	`VInterrupt`	Enters the INTERRUPT state
_NTO_TRACE_VTHJOIN	`VTHJOIN`	`VJoin`	Enters the JOIN state
_NTO_TRACE_VTHMUTEX	`VTHMUTEX`	`VMutex`	Enters the MUTEX state
_NTO_TRACE_VTHNANOSLEEP	`VTHNANOSLEEP`	`VNanosleep`	Enters the NANOSLEEP state
_NTO_TRACE_VTHNET_REPLY	`VTHNET_REPLY`	`VNetReply`	Enters the NET_REPLY state
_NTO_TRACE_VTHNET_SEND	`VTHNET_SEND`	`VNetSend`	Enters the NET_SEND state
_NTO_TRACE_VTHREADY	`VTHREADY`	`VReady`	Enters the READY state
_NTO_TRACE_VTHRECEIVE	`VTHRECEIVE`	`VReceive`	Enters the RECEIVE state
_NTO_TRACE_VTHREPLY	`VTHREPLY`	`VReply`	Enters the REPLY state

Event	traceprinter label	IDE label	Emitted when a virtual thread:
_NTO_TRACE_VTHRUNNING	VTHRUNNING	VRunning	Enters the RUNNING state
_NTO_TRACE_VTHSEM	VTHSEM	VSemaphore	Enters the SEM state
_NTO_TRACE_VTHSEND	VTHSEND	VSend	Enters the SEND state
_NTO_TRACE_VTHSIGSUSPEND	VTHSIGSUSPEND	VSigSuspend	Enters the SIGSUSPEND state
_NTO_TRACE_VTHSIGWAITINFO	VTHSIGWAITINFO	VSigWaitInfo	Enters the SIGWAITINFO state
_NTO_TRACE_VTHSTACK	VTHSTACK	VStack	Enters the STACK state
_NTO_TRACE_VTHSTOPPED	VTHSTOPPED	VStopped	Enters the STOPPED state
_NTO_TRACE_VTHWAITCTX	VTHWAITCTX	VWaitCtx	Enters the WAITCTX state
_NTO_TRACE_VTHWAITPAGE	VTHWAITPAGE	VWaitPage	Enters the WAITPAGE state
_NTO_TRACE_VTHWAITTHREAD	VTHWAITTHREAD	VWaitThread	Enters the WAITTHREAD state

Chapter 3
Kernel Buffer Management

As the instrumented kernel intercepts events, it stores them in a ring of buffers.

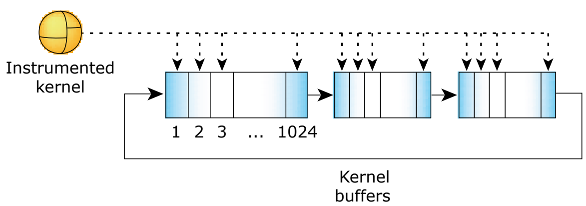

Figure 4: The kernel buffers.

As each buffer fills, the instrumented kernel raises an _NTO_HOOK_TRACE synthetic interrupt to notify the data-capturing program that the buffer is ready to be read.

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 KB

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 *Interpreting Trace Data* chapter.

Ring buffer 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 KB 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 in contiguous physical memory space. If the data-capture program requests a larger block than is available contiguously, the instrumented kernel returns 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 ring 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, the instrumented kernel raises an _NTO_HOOK_TRACE synthetic interrupt to notify the data-capturing program to save the buffer. Because the buffer size is fixed, the kernel sends only the buffer index; 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:

- around 70% (_TRACEBUF_MAX_EVENTS) for linear mode
- around 95% (_TRACEBUF_MAX_EVENTS_RING) for ring mode

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 that the buffers be flushed more often. The data-capture program handles each 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 raises an _NTO_HOOK_TRACE synthetic interrupt to notify 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 *Tutorials* chapter.

Chapter 4
Capturing Trace Data

The program that captures data is the “messenger” between the instrumented kernel and the filesystem.

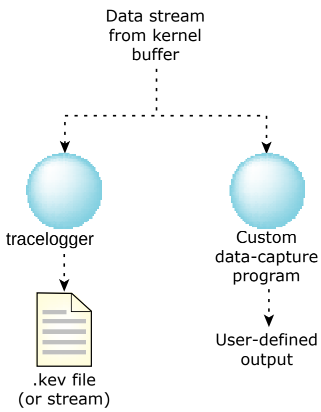

Figure 5: Possible data capture configurations.

The main function of the data-capture program is to send 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 program must also:

- interface with the instrumented kernel
- specify data-filtering requirements the instrumented kernel will use

You must configure the instrumented kernel before logging. The instrumented kernel configuration settings include:

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

The instrumented kernel retains the settings, and multiple programs access a single instrumented kernel configuration. Changing the settings in one process supersedes the settings made in another.

We've provided `tracelogger` as the default data-capture utility. Although you can write your own utility, there's little need to.

You can control the capture of data via `qconn` (under the control of the IDE), `tracelogger` (from the command line), or directly from your application. All three approaches use the *TraceEvent()* function to control the instrumented kernel:

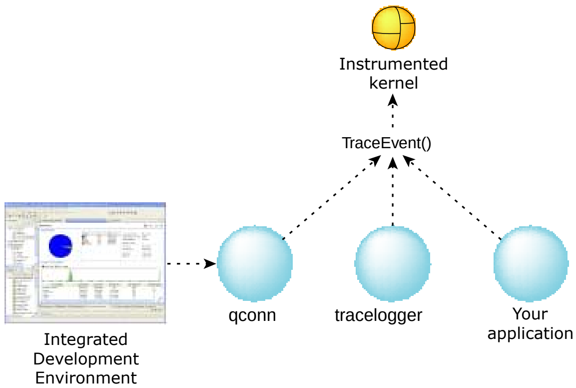

Figure 6: Controlling the capture of trace data.

For information about controlling the trace from the IDE, see the “Analyzing System Behavior” chapter of the IDE *User's Guide*.

Let's look first at using `tracelogger`, and then we'll describe how you can use *TraceEvent()* to control tracing from your application.

CAUTION:

- Don't run more than one instance of `tracelogger` at a time. Similarly, don't run `tracelogger` and trace events under control of the IDE at the same time.
- In QNX Neutrino 7.0 or later, we require that the hardware underlying *ClockCycles()* be synchronized across all processors on an SMP system. If the clocks aren't synchronized, then `tracelogger` produces data with inconsistent timestamps, and the IDE won't be able to load the trace file. The IDE attempts to properly order the events in the trace file, and this can go awry if the timestamp data is incorrect.

  The `traceprinter` utility doesn't have any issues with such traces because it doesn't attempt to reorder the data and interpret it; it simply dumps the contents of each event.

Using tracelogger to control tracing

The options that you use when you start tracelogger affect the way that the instrumented kernel logs events *and* how tracelogger captures them.

Managing trace buffers

You can use tracelogger's command-line options to manage the instrumented kernel's buffers.

You can specify:

- the number of buffers
- whether or not to preserve the buffers in shared memory, to reuse later

You can also specify the number of buffers that tracelogger itself uses.

For more information, see the entry for tracelogger in the *Utilities Reference*.

tracelogger's modes of operation

You can run tracelogger in several modes—depending on how and what you want to trace—by specifying the following command-line options:

Mode	Option	The kernel:	tracelogger:
Continuous	-c	Logs events	Captures the events, and continues to do so until terminated
Daemon	-d1	Doesn't log events[a]	Waits passively[a]
Iterations (the default)	-n	Logs events	Captures *num_buffers* of data and then terminates
Ring	-r	Logs events	Doesn't capture events until it gets a SIGINT signal, or an application calls *TraceEvent()*[b] with a command of _NTO_TRACE_STOP
Time-based	-s	Logs events	Captures events for the specified number of seconds

[a] In daemon mode, logging starts when an application calls TraceEvent(_NTO_TRACE_START) and continues until an application calls TraceEvent(_NTO_TRACE_STOP), or until you terminate tracelogger.

[b] When you terminate tracing in ring mode, tracelogger stops logging events, and then briefly restarts and stops it again so it can capture the state information that's emitted by the _NTO_TRACE_START command. This information includes the thread IDs and names of processes.

In the non-daemon modes, you configure, start, and stop the tracing from the command line. In daemon mode, your application must do everything from code, but if you also specify the `-E` option, `tracelogger` enables all events in all classes, and you can use the `-F` option to set up filtering.

Here's an outline of the strengths, weaknesses, and features of these modes:

Feature	Non-daemon mode	Daemon mode	Daemon mode with -E
`tracelogger` **support**	Full	Limited	Full
Controllability	Limited	Full	Full
Events recorded by default	All	None	All
Configuration difficulty	Easy	Harder	Easy
Configuration method	Command line only	User program, using calls to *TraceEvent()*	Command line and user program
Logging starts	Instantaneously	User program, using calls to *TraceEvent()*	User program, using calls to *TraceEvent()*

If an application has called `TraceEvent(_NTO_TRACE_START)`, and you then try to start `tracelogger`, `tracelogger` might fail with a "resource busy" message. To help avoid this:

- Start `tracelogger` before your application issues a _NTO_TRACE_START or _NTO_TRACE_STARTNOSTATE command.
- Don't leave tracing on indefinitely; be sure to issue a _NTO_TRACE_STOP after each _NTO_TRACE_START or _NTO_TRACE_STARTNOSTATE command.

For a full description of the `tracelogger` utility and its options, see its entry in the *Utilities Reference*.

Choosing between wide and fast modes

By default, the instrumented kernel and `tracelogger` collect data in fast mode; to switch to wide mode, specify the `-w` option when you start `tracelogger`.

Filtering events

The `tracelogger` utility gives you some basic control over filtering by way of its `-F` option. This filtering is limited to excluding entire classes of events at a time; if you need a finer granularity, you'll need to use *TraceEvent()*, as described in the *Filtering* chapter in this guide.

By default, `tracelogger` captures all events from all classes, but you can disable the tracing of events from the classes as follows:

To disable this class:	Specify:
Kernel calls	`-F1`
Interrupt	`-F2`
Process	`-F3`
Thread	`-F4`
Virtual thread	`-F5`
Communication	`-F6`
System	`-F7`

You can specify more than one filter by using multiple `-F` options. Note that you can't disable the Control or User classes with this option. For more information about classes, see the *Events and the Kernel* chapter of this guide.

Specifying where to send the output

Because the ring 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 pipes the information to either an output device or a file.

By default, the `tracelogger` utility saves the output in the binary file **/dev/shmem/tracebuffer.kev**, but you can use the `-f` option to specify a different path. The `.kev` extension is short for “kernel events”; you can use a different extension, but the IDE recognizes `.kev` and automatically uses the System Profiler to open such files.

You can also map the file in shared memory (`-M`), but you must then also specify the maximum size for the file (`-S`).

Using *TraceEvent()* to control tracing

You don't have to use `tracelogger` to control all aspects of tracing; you can call *TraceEvent()* directly—which (after all) is what `tracelogger` does. Using *TraceEvent()* to control tracing means a bit more work for you, but you have much more control over specific details.

You could decide not to use `tracelogger` at all, and use *TraceEvent()* exclusively, but you'd then have to manage the buffers, collect the trace data, and save it in the appropriate form, which would be a significant amount of work.

In practical terms you'll likely use `tracelogger` and *TraceEvent()* together. For example, you might run `tracelogger` in daemon mode, to take advantage of its management of the trace data, but call *TraceEvent()* to control exactly which events to trace.

The *TraceEvent()* kernel call takes a variable number of arguments. The first is always a command and determines what (if any) additional arguments are required. For reference information about *TraceEvent()*, see the QNX Neutrino *C Library Reference*.

Managing trace buffers

As mentioned above, you can use *TraceEvent()* to manage the instrumented kernel's buffers, but it's probably easier to run `tracelogger` in daemon mode and let it look after the buffers. Nevertheless, here's a summary of how to do it with *TraceEvent()*:

In order to allocate or free the trace buffers, your application must have the PROCMGR_AID_TRACE ability enabled. For more information, see the entry *procmgr_ability()* in the QNX Neutrino *C Library Reference*.

- To allocate the buffers, use the _NTO_TRACE_ALLOCBUFFER command, specifying the number of buffers and a pointer to a location where *TraceEvent()* can store the physical address of the beginning of the ring of allocated trace buffers:

  ```
  TraceEvent(_NTO_TRACE_ALLOCBUFFER, uint bufnum, void** paddr);
  ```

  Allocated trace buffers can store 1024 simple trace events.

  Once you've allocated the buffers, you can use *mmap_device_memory()* to map the buffers into your process's address space and get their virtual address. For example:

  ```
  paddr_t paddr;
  tracebuf_t *kbufs;

  ret = TraceEvent(_NTO_TRACE_ALLOCBUFFER, num_buffers, &paddr);
  if( ret == -1 )
  {
    // Handle the error.
  }
  ```



```
kbufs = mmap_device_memory(0, num_buffers * sizeof(tracebuf_t), PROT_READ |
                           0, paddr );
if( kbufs == MAP_FAILED )
{
  // Handle the error.
}
```

Then you can use *InterruptHookTrace()* to register a handler for the _NTO_HOOK_TRACE synthetic interrupt that the instrumented kernel raises as each buffer becomes full.

- To free the buffers, use the _NTO_TRACE_DEALLOCBUFFER command. It doesn't take any additional arguments:

  ```
  TraceEvent(_NTO_TRACE_DEALLOCBUFFER);
  ```

  All events stored in the trace buffers are lost.

- To flush the buffer, regardless of the number of trace events it contains, use the _NTO_TRACE_FLUSHBUFFER command:

  ```
  TraceEvent(_NTO_TRACE_FLUSHBUFFER);
  ```

- To get the number of simple trace events that are currently stored in the trace buffer, use the _NTO_TRACE_QUERYEVENTS command:

  ```
  num_events = TraceEvent(_NTO_TRACE_QUERYEVENTS);
  ```

For examples of some of these commands, see "*Data-capture program*" in the "*Sample Programs*" appendix.

Modes of operation

TraceEvent() doesn't support the different modes of operation that `tracelogger` does; your application has to indicate when to start tracing, how long to trace for, and so on:

- To choose linear or ring mode, use the _NTO_TRACE_SETLINEARMODE or _NTO_TRACE_SETRINGMODE command:

  ```
  TraceEvent(_NTO_TRACE_SETLINEARMODE);
  TraceEvent(_NTO_TRACE_SETRINGMODE);
  ```

  As described earlier in this chapter, in ring mode the kernel stores all events in the ring of buffers without flushing them. In linear mode (the default), every filled-up buffer is captured and flushed immediately.

  _NTO_TRACE_SETLINEARMODE and _NTO_TRACE_SETRINGMODE cause the trace buffers to be cleared, so you should use these commands before you start tracing.

- To start tracing, use the _NTO_TRACE_START or _NTO_TRACE_STARTNOSTATE command:

  ```
  TraceEvent(_NTO_TRACE_START);
  TraceEvent(_NTO_TRACE_STARTNOSTATE);
  ```

These commands are similar, except that _NTO_TRACE_STARTNOSTATE suppresses the initial system state information (which includes thread IDs and the names of processes). This information is overwritten when the kernel reuses the buffer; if you're logging events in ring mode, you can make sure you capture the process names by issuing an _NTO_TRACE_START command followed by _NTO_TRACE_STOP after you've finished tracing.

- To stop tracing, use the _NTO_TRACE_STOP command:

```
TraceEvent(_NTO_TRACE_STOP);
```

You can decide whether to trace until you've gathered a certain quantity of data, trace for a certain length of time, or trace only during an operation that's of particular interest to you. After stopping the trace, you should flush the buffer by calling:

```
TraceEvent(_NTO_TRACE_FLUSHBUFFER);
```

If an application has called `TraceEvent(_NTO_TRACE_START)`, and you then try to start `tracelogger`, `tracelogger` might fail with a “resource busy” message. To help avoid this:

- Start `tracelogger` before your application issues a _NTO_TRACE_START or _NTO_TRACE_STARTNOSTATE command.
- Don't leave tracing on indefinitely; be sure to issue a _NTO_TRACE_STOP after each _NTO_TRACE_START or _NTO_TRACE_STARTNOSTATE command.

Filtering events

You can select events in an additive or subtractive manner; you can start with no events, and then add specific classes or events, or you can start with all events, and then exclude specific ones. We'll discuss using *TraceEvent()* to filter events in the *Filtering* chapter.

Choosing between wide and fast modes

TraceEvent() gives you much finer control over wide and fast mode than you can get with `tracelogger`, which can simply set the mode for *all* events in *all* traced classes. Using *TraceEvent()*, you can set fast and wide mode for all classes, a specific class, or a specific event in a class:

- To set the mode for all classes, use the _NTO_TRACE_SETALLCLASSESWIDE or _NTO_TRACE_SETALLCLASSESFAST command. These commands don't require any additional arguments:

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

- To set the mode for all events in a class, use the _NTO_TRACE_SETCLASSFAST or _NTO_TRACE_SETCLASSWIDE command. These commands require a class as an additional argument:

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

  For example:

  ```
  TraceEvent(_NTO_TRACE_SETCLASSWIDE, _NTO_TRACE_KERCALLENTER);
  ```

- To set the mode for a specific event in a class, use the _NTO_TRACE_SETEVENTFAST or _NTO_TRACE_SETEVENTWIDE command, specifying the class, followed by the event:

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

  For example:

  ```
  TraceEvent(_NTO_TRACE_SETEVENTFAST, _NTO_TRACE_KERCALLENTER,
             __KER_INTERRUPT_ATTACH);
  ```

Inserting trace events

You can even use *TraceEvent()* to insert your own events into the trace data. You can call *TraceEvent()* directly (see below), but it's much easier to use the following convenience functions:

trace_func_enter()

Insert a trace event for the entry to a function

trace_func_exit()

Insert a trace event for the exit from a function

trace_here()

Insert a trace event for the current address

trace_logb()

Insert a user combine trace event

trace_logbc()

Insert a trace event of an arbitrary class and type with arbitrary data

trace_logf()

Insert a user string trace event

trace_logi()

Insert a user simple trace event

trace_nlogf()

Insert a user string trace event, specifying a maximum string length

trace_vnlogf()

Insert a user string trace event, using a variable argument list

If you want to call *TraceEvent()* directly, use one of the following commands:

- _NTO_TRACE_INSERTCCLASSEVENT
- _NTO_TRACE_INSERTCUSEREVENT
- _NTO_TRACE_INSERTEVENT
- _NTO_TRACE_INSERTSCLASSEVENT
- _NTO_TRACE_INSERTSUSEREVENT
- _NTO_TRACE_INSERTUSRSTREVENT

For more information, see the entry for *TraceEvent()* in the QNX Neutrino *C Library Reference*.

Chapter 5
Filtering

Gathering many events generates a lot of data, which requires *memory* and *processor time*. It also makes the task of interpreting the data more difficult.

Because the amount of data that the instrumented kernel generates can be overwhelming, the SAT supports several types of *filters* that you can use to reduce the amount of data to be processed:

Static rules filter

A simple filter that chooses events based on their type, class, or other simple criteria.

Dynamic rules filter

A more complex filter that lets you register a callback function that can decide—based on the state of your application or system, or on whatever criteria you choose—whether or not to log a given event.

Post-processing filter

A filter that you run after capturing event data. Like the dynamic rules filter, this can be as complex and sophisticated as you wish.

The static and dynamic rules filters affect the amount of data being logged into the kernel buffers; filtered data is discarded—you save processing time and memory, but there's a chance that some of the filtered data could have been useful.

In contrast, the post-processing facility doesn't discard data; it simply doesn't use it—if you've saved the data, you can use it later.

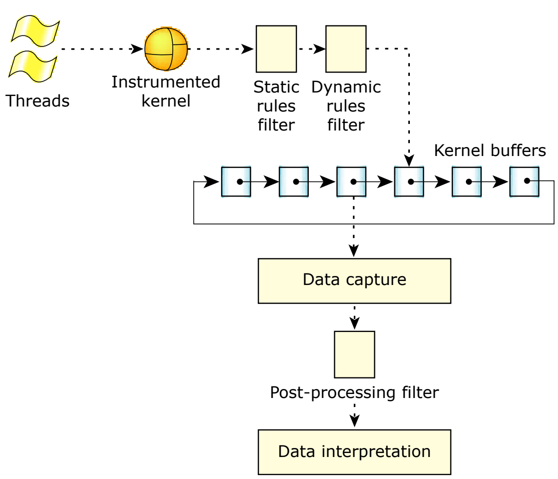

Figure 7: Overall view of the SAT and its filters.

Most of the events don't indicate what caused the event to occur. For example, an event for entering *MsgSendv()* doesn't indicate which thread in which process called it; you have to infer it during interpretation from a previous thread-running event. You have carefully choose what you filter to avoid losing this context.

The static rules filter

You can use the static rules filter to track or filter events for all classes, certain events in a class, or even events related to specific process and thread IDs. You can select events in an additive or subtractive manner; you can start with no events and then add specific classes or events, or you can start with all events and then exclude specific ones.

The static rules filter is the best, most efficient method of data reduction. It generally frees up the processor while significantly reducing the data rate. This 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.

You set up this filter using various *TraceEvent()* commands. For information about the different classes, see "*Classes and events*" in the "Events and the Kernel" chapter of this guide.

You can set up process- and thread-specific tracing (using the _NTO_TRACE_SET* and _NTO_TRACE_CLR* commands described below) only for the following classes:

- _NTO_TRACE_COMM
- _NTO_TRACE_KERCALL, _NTO_TRACE_KERCALLENTER, _NTO_TRACE_KERCALLEXIT
- _NTO_TRACE_SYSTEM
- _NTO_TRACE_THREAD
- _NTO_TRACE_VTHREAD

Here's a general outline for using the static rules filter:

1. Clear any existing filters. The instrumented kernel retains its settings, so you should be careful not to make any assumptions about the settings that are in effect when you set up your filters. Start by removing the tracing for *all* classes and events:

   ```
   TraceEvent(_NTO_TRACE_DELALLCLASSES);
   ```

   The _NTO_TRACE_DELALLCLASSES command doesn't suppress any process- and thread-specific tracing that might have previously been set up. You need to clear it separately, by using the following *TraceEvent()* commands:

To clear:	Call ***TraceEvent()*** with these arguments:
Process-specific tracing of all events of a given class	`_NTO_TRACE_CLRCLASSPID, int` *class*
Process- and thread-specific tracing of all events of a given class	`_NTO_TRACE_CLRCLASSTID, int` *class*
Process-specific tracing of the given event in a given class	`_NTO_TRACE_CLREVENTPID, int` *class*, `int` *event*
Process- and thread-specific tracing of a given event and class	`_NTO_TRACE_CLREVENTTID, int` *class*, `int` *event*

For example, you might want to start by turning off all filtering:

```
TraceEvent(_NTO_TRACE_DELALLCLASSES);
TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_KERCALL);
TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_KERCALL);
TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_THREAD);
TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_THREAD);
TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_VTHREAD);
TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_VTHREAD);
TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_SYSTEM);
TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_SYSTEM);
TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_COMM);
TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_COMM);
```

2. Set up the tracing of the event classes that you're interested in. You can do this in an additive or subtractive manner; you can start with no events, and then add specific classes or events, or you can start with all events, and then exclude specific ones.

To:	Call *TraceEvent()* with these arguments:
Enable the tracing of *all* classes and events	`_NTO_TRACE_ADDALLCLASSES`
Enable the tracing of all events in a specific class	`_NTO_TRACE_ADDCLASS,` *class*
Enable the tracing of a specific event in a specific class	`_NTO_TRACE_ADDEVENT,` *class*, *event*
Disable the tracing of all events in a specific class	`_NTO_TRACE_DELCLASS,` *class*
Disable the tracing of a specific event in a specific class	`_NTO_TRACE_DELEVENT,` *class*, *event*

3. Optionally restrict the tracing to a specific process, thread, or both:

To trace:	Call *TraceEvent()* with these arguments:
All events for the given class that are for the process with the given ID	`_NTO_TRACE_SETCLASSPID, int` *class*, `pid_t` *pid*
All events for the given class that are for the process and thread with the given IDs	`_NTO_TRACE_SETCLASSTID, int` *class*, `pid_t` *pid*, `uint32_t` *tid*
All events of the given type in the given class that are for the process with the given ID	`_NTO_TRACE_SETEVENTPID, int` *class*, `int` *event*, `pid_t` *pid*
All events of the given type in the given class that are for the process and thread with the given IDs	`_NTO_TRACE_SETEVENTTID, int` *class*, `int` *event*, `pid_t` *pid*, `uint32_t` *tid*

You can set up class or event filtering for one process or thread at a time. For example, the following sets up filtering for different classes for different processes:

```
TraceEvent(_NTO_TRACE_SETCLASSPID, _NTO_TRACE_KERCALL, pid_1);
TraceEvent(_NTO_TRACE_SETCLASSTID, _NTO_TRACE_THREAD, pid_2, tid_1);
```

but the second call in the following overrides the setting made in the first call:

```
TraceEvent(_NTO_TRACE_SETCLASSPID, _NTO_TRACE_KERCALL, pid_1);
TraceEvent(_NTO_TRACE_SETCLASSTID, _NTO_TRACE_KERCALL, pid_2, tid_1);
```

For an example that uses the static filter, see the *five_events.c* example in the "*Tutorials*" 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 quick. This filter lets you register functions (event handlers) that decide whether or not to log a given event.

If you want to use a dynamic rules filter, be sure that you've also set up a static rules filter that logs the events you want to examine. For example, if you want to dynamically examine events in the _NTO_TRACE_THREAD class, also call:

```
TraceEvent(_NTO_TRACE_ADDCLASS, _NTO_TRACE_THREAD):
```

For an example of using the dynamic rules filter, see the *eh_simple.c* example in the *Tutorials* chapter.

Setting up a dynamic rules filter

Before you set up dynamic filtering, you must:

- have the PROCMGR_AID_TRACE and PROCMGR_AID_IO abilities enabled. For more information, see the entry for *procmgr_ability()* in the QNX Neutrino *C Library Reference*.
- request I/O privileges by calling *ThreadCtl()* with the _NTO_TCTL_IO flag:

```
if (ThreadCtl(_NTO_TCTL_IO, 0)!=EOK) {
   fprintf(stderr, "argv[0]: Failed to obtain I/O privileges\n");
   return (-1);
}
```

Then call *TraceEvent()* with one of these commands:

_NTO_TRACE_ADDCLASSEVHANDLER

Register a function to call whenever an event for the given class is emitted:

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

_NTO_TRACE_ADDEVENTHANDLER

Register a function to call whenever an event for the given class and event type is emitted:

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

The additional arguments are:

event_hdlr

A pointer to the function that you want to register. The prototype for the function is:

```
int event_hdlr (event_data_t *event_data);
```

data_struct

A pointer to a locally defined data structure, of type `event_data_t`, where the kernel can store event data to pass to the event handler (see below).

Event handler

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.

CAUTION: The only library functions that you can call in your event handler are those that are safe to call from an interrupt handler. For a list of these functions, see the Full Safety Information appendix in the QNX Neutrino *C Library Reference*. If you call an unsafe function—such as *printf()*—in your event handler, you'll crash your entire system. Your event handler must also be reentrant.

If you want to log the current event, return a non-zero value; to discard the event, return 0. Here's a very simple event handler that says to log all of the given events:

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

If you use both types of dynamic filters (event handler and class event handler), and they both apply to a particular event, the event is logged if *both* event handlers return a non-zero value.

In addition to deciding whether or not the event should be logged, you can use the dynamic rules filter 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.

You can access the information about the intercepted event within the event handler by examining the `event_data_t` structure passed as an argument to the event handler. The layout of the `event_data_t` structure (declared in **<sys/trace.h>**) is as follows:

```
/* event data filled by an event handler */
typedef struct
{
   __traceentry header;      /* same as traceevent header    */
   uint32_t*    data_array;  /* initialized by the user      */
   uint32_t     el_num;      /* number of elements returned  */
   void*        area;        /* user data                    */
   uint32_t     feature_mask;/* bits indicate valid features */
```

```
    uint32_t      feature[_NTO_TRACE_FI_NUM]; /* feature array
                                               - additional data */
} event_data_t;
```

CAUTION: The `event_data_t` structure includes a pointer to an array for the data arguments of the event. You *must* provide an array, and it must be large enough to hold the data for the event or events that you're handling (see the *Current Trace Events and Data* appendix). For example:

```
event_data_t e_d_1;
uint32_t     data_array_1[20]; /* 20 elements for potential args. */

e_d_1.data_array = data_array_1;
```

If you don't provide the data array, or it isn't big enough, your data segment could become corrupted.

You can use the following macros, defined in **<sys/trace.h>**, to work with the header of an event:

_NTO_TRACE_GETEVENT_C(c)

Get the class.

_NTO_TRACE_GETEVENT(c)

Get the type of event.

_NTO_TRACE_GETCPU(h)

Get the number of the CPU that the event occurred on.

_NTO_TRACE_SETEVENT_C(c,cl)

Set the class in the header *c* to be *cl*.

_NTO_TRACE_SETEVENT(c, e)

Set the event type in the header *c* to be *e*.

The bits of the *feature_mask* member are related to any additional features (arguments) that you can access 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 the *data_array*.

There are two constants associated with each additional feature:

- `_NTO_TRACE_FM***` — feature parameter masks
- `_NTO_TRACE_FI***` — feature index parameters

The currently defined features are:

Feature	Parameter mask	Index
Process ID	_NTO_TRACE_FMPID	_NTO_TRACE_FIPID
Thread ID	_NTO_TRACE_FMTID	_NTO_TRACE_FITID

If any particular bit of the *feature_mask* is set to 1, then you can access the feature corresponding to this bit within the *feature* array. Otherwise, you must not access the feature. For example, if the expression:

```
feature_mask & _NTO_TRACE_FMPID
```

is TRUE, then you can access the additional feature corresponding to identifier _NTO_TRACE_FMPID as:

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

Removing event handlers

To remove event handlers, call *TraceEvent()* with these commands:

_NTO_TRACE_DELCLASSEVHANDLER

Remove the function for the given class and event type:

```
TraceEvent(_NTO_TRACE_DELCLASSEVHANDLER, class);
```

_NTO_TRACE_DELEVENTHANDLER

Remove the function for the given class and event type:

```
TraceEvent(_NTO_TRACE_DELEVENTHANDLER, class, event);
```

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 is 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. One example of post-processing is the `traceprinter` utility itself. It prints all the events instead of filtering them, but the principles are the same.

We'll look at `traceprinter` in more detail in the *Interpreting Trace Data* chapter.

Chapter 6
Interpreting Trace Data

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

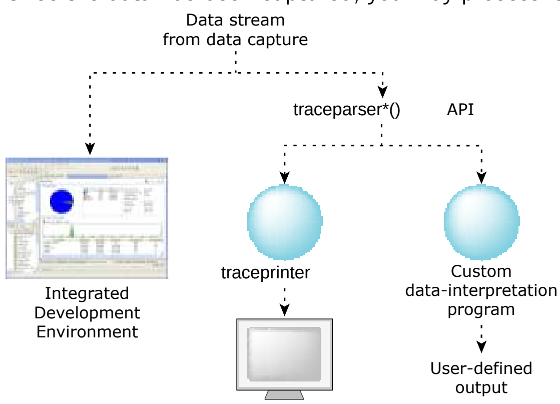

Figure 8: Possible data interpretation configurations.

The best tool (by far) for interpreting the copious amounts of trace data is the Integrated Development Environment. It provides a sophisticated and versatile user interface that lets you filter and examine the data. For more information, see the "Analyzing System Behavior" chapter of the IDE *User's Guide*.

We also provide a `traceprinter` utility that simply prints a plain-text version of the trace data, sending its output to *stdout* or to a file.

You can also build your own, custom interpreter, using the **traceparser** library.

Using **`traceprinter`** and interpreting the output

The simplest way to turn the tracing data into a form that you can analyze is to pass the **.kev** file through `traceprinter`. For details, see its entry in the *Utilities Reference*.

Let's take a look at an example of the output from `traceprinter`. This is the output from "*Gathering all events from all classes*" in the Tutorials chapter.

The output starts with some information about how you ran the trace:

```
TRACEPRINTER version 1.02
TRACEPARSER LIBRARY version 1.02
 -- HEADER FILE INFORMATION --
       TRACE_FILE_NAME:: all_classes.kev
            TRACE_DATE:: Wed Jun 24 10:52:58 2009
       TRACE_VER_MAJOR:: 1
       TRACE_VER_MINOR:: 01
   TRACE_LITTLE_ENDIAN:: TRUE
        TRACE_ENCODING:: 16 byte events
       TRACE_BOOT_DATE:: Tue Jun 23 11:47:46 2009
  TRACE_CYCLES_PER_SEC:: 736629000
         TRACE_CPU_NUM:: 1
         TRACE_SYSNAME:: QNX
        TRACE_NODENAME:: localhost
     TRACE_SYS_RELEASE:: 6.4.1
     TRACE_SYS_VERSION:: 2009/05/20-17:35:56EDT
         TRACE_MACHINE:: x86pc
     TRACE_SYSPAGE_LEN:: 2264
TRACE_TRACELOGGER_ARGS:: tracelogger -d1 -n 3 -f all_classes.kev
```

The next section includes information about all the processes in existence when the trace started:

```
 -- KERNEL EVENTS --
t:0x4f81e320 CPU:00 CONTROL: BUFFER sequence = 33, num_events = 714
t:0x4f81e320 CPU:00 CONTROL :TIME msb:0x000037b0 lsb(offset):0x4f81e014
t:0x4f82017a CPU:00 PROCESS :PROCCREATE_NAME
                      ppid:0
                       pid:1
                      name:proc/boot/procnto-smp-instr
t:0x4f820f9a CPU:00 THREAD  :THCREATE      pid:1 tid:1
t:0x4f821358 CPU:00 THREAD  :THREADY       pid:1 tid:1
t:0x4f821698 CPU:00 THREAD  :THCREATE      pid:1 tid:2
t:0x4f821787 CPU:00 THREAD  :THRECEIVE     pid:1 tid:2
t:0x4f8219ca CPU:00 THREAD  :THCREATE      pid:1 tid:3
t:0x4f821ac6 CPU:00 THREAD  :THRECEIVE     pid:1 tid:3
t:0x4f821c94 CPU:00 THREAD  :THCREATE      pid:1 tid:4
t:0x4f821d90 CPU:00 THREAD  :THRECEIVE     pid:1 tid:4
t:0x4f821f6c CPU:00 THREAD  :THCREATE      pid:1 tid:5
t:0x4f82205b CPU:00 THREAD  :THRECEIVE     pid:1 tid:5
t:0x4f8222aa CPU:00 THREAD  :THCREATE      pid:1 tid:7
t:0x4f822399 CPU:00 THREAD  :THRECEIVE     pid:1 tid:7
t:0x4f8225bd CPU:00 THREAD  :THCREATE      pid:1 tid:8
t:0x4f8226ac CPU:00 THREAD  :THRECEIVE     pid:1 tid:8
t:0x4f8228ca CPU:00 THREAD  :THCREATE      pid:1 tid:10
t:0x4f8229b9 CPU:00 THREAD  :THRECEIVE     pid:1 tid:10
t:0x4f822b7d CPU:00 THREAD  :THCREATE      pid:1 tid:11
t:0x4f822c6c CPU:00 THREAD  :THRECEIVE     pid:1 tid:11
t:0x4f822dd7 CPU:00 THREAD  :THCREATE      pid:1 tid:12
t:0x4f822ec6 CPU:00 THREAD  :THRECEIVE     pid:1 tid:12
t:0x4f8230ac CPU:00 THREAD  :THCREATE      pid:1 tid:15
t:0x4f82319b CPU:00 THREAD  :THRECEIVE     pid:1 tid:15
t:0x4f8233ca CPU:00 THREAD  :THCREATE      pid:1 tid:20
t:0x4f8234b9 CPU:00 THREAD  :THRECEIVE     pid:1 tid:20
t:0x4f823ad0 CPU:00 PROCESS :PROCCREATE_NAME
                      ppid:1
                       pid:2
                      name:sbin/tinit
t:0x4f823f38 CPU:00 THREAD  :THCREATE      pid:2 tid:1
```



```
t:0x4f82402e CPU:00 THREAD  :THREPLY       pid:2 tid:1
t:0x4f82447d CPU:00 PROCESS :PROCCREATE_NAME
                      ppid:2
                       pid:4099
                      name:proc/boot/pci-server
t:0x4f824957 CPU:00 THREAD  :THCREATE      pid:4099 tid:1
t:0x4f824a4d CPU:00 THREAD  :THRECEIVE     pid:4099 tid:1
t:0x4f824ff8 CPU:00 PROCESS :PROCCREATE_NAME
                      ppid:2
                       pid:4100
                      name:proc/boot/slogger2
```

You can suppress this initial information by passing the _NTO_TRACE_STARTNOSTATE command to *TraceEvent()*, but you'll likely need the information (including process IDs and thread IDs) to make sense out of the actual trace data.

The sample above shows the creation and naming of the instrumented kernel `procnto-smp-instr` (process ID 1) and its threads (thread ID 1 is the idle thread), followed by `tinit` (process ID 2), `pci-server`, and `slogger2`. Some of these are the processes that were launched when you booted your system.

This continues for a while, culminating in the creation of the `tracelogger` process and our own program, `all_classes` (process ID 1511472):

```
t:0x4f852aa8 CPU:00 PROCESS :PROCCREATE_NAME
                      ppid:426015
                       pid:1507375
                      name:usr/sbin/tracelogger
t:0x4f853360 CPU:00 THREAD  :THCREATE      pid:1507375 tid:1
t:0x4f853579 CPU:00 THREAD  :THRECEIVE     pid:1507375 tid:1
t:0x4f85392a CPU:00 THREAD  :THCREATE      pid:1507375 tid:2
t:0x4f853a19 CPU:00 THREAD  :THSIGWAITINFO pid:1507375 tid:2
t:0x4f853d96 CPU:00 PROCESS :PROCCREATE_NAME
                      ppid:426022
                       pid:1511472
                      name:./all_classes
t:0x4f854048 CPU:00 THREAD  :THCREATE      pid:1511472 tid:1
t:0x4f854140 CPU:00 THREAD  :THRUNNING     pid:1511472 tid:1
```

Next is the exit from our program's call to *TraceEvent()*:

```
t:0x4f854910 CPU:00 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000 empty:0x00000000
```

Why doesn't the trace doesn't include the *entry* to *TraceEvent()*? Well, `tracelogger` didn't log anything until our program *told* it to—by calling *TraceEvent()*!

So far, so good, but now things get more complicated:

```
t:0x4f856aac CPU:00 KER_CALL:THREAD_DESTROY/47 tid:-1 status_p:0
t:0x4f857dca CPU:00 KER_EXIT:THREAD_DESTROY/47 ret_val:0x00000030 empty:0x00000000
t:0x4f8588d3 CPU:00 KER_CALL:THREAD_DESTROYALL/48 empty:0x00000000 empty:0x00000000
t:0x4f858ed7 CPU:00 THREAD  :THDESTROY     pid:1511472 tid:1
t:0x4f8598b9 CPU:00 THREAD  :THDEAD        pid:1511472 tid:1
t:0x4f859c4c CPU:00 THREAD  :THRUNNING     pid:1 tid:1
```

You can see that a thread is being destroyed, but which one? The *tid* of -1 refers to the current thread, but which process does it belong to? As mentioned earlier, most of the events don't indicate what caused the event to occur; you have to infer from a previous thread-running event. In this case, it's our own program (process ID 1511472) that's ending; it starts the tracing, and then exits. Thread 1 of `procnto-smp-instr` (the idle thread) runs.

The trace continues like this:

```
t:0x4f85c6e3 CPU:00 COMM    :SND_PULSE_EXE scoid:0x40000002 pid:1
t:0x4f85cecd CPU:00 THREAD  :THRUNNING     pid:1 tid:12
t:0x4f85d5ad CPU:00 THREAD  :THREADY       pid:1 tid:1
```

```
t:0x4f85e5b3 CPU:00 COMM    :REC_PULSE     scoid:0x40000002 pid:1
t:0x4f860ee2 CPU:00 KER_CALL:THREAD_CREATE/46 func_p:f0023170 arg_p:eff6e000
t:0x4f8624c7 CPU:00 THREAD  :THCREATE      pid:1511472 tid:1
t:0x4f8625ff CPU:00 THREAD  :THWAITTHREAD  pid:1 tid:12
t:0x4f8627b4 CPU:00 THREAD  :THRUNNING     pid:1511472 tid:1
t:0x4f8636fd CPU:00 THREAD  :THREADY       pid:1 tid:12
t:0x4f865c34 CPU:00 KER_CALL:CONNECT_SERVER_INFO/41 pid:0 coid:0x00000000
t:0x4f866836 CPU:00 KER_EXIT:CONNECT_SERVER_INFO/41 coid:0x00000000 info->nd:0
t:0x4f86735e CPU:00 KER_CALL:TIMER_TIMEOUT/75 timeout_flags:0x00000050 ntime(sec):30
t:0x4f868445 CPU:00 KER_EXIT:TIMER_TIMEOUT/75 prev_timeout_flags:0x00000000 otime(sec):0
t:0x4f8697d3 CPU:00 INT_ENTR:0x00000000 (0)       IP:0xf008433e
t:0x4f86a276 CPU:00 INT_HANDLER_ENTR:0x00000000 (0)       PID:126997 IP:0x080b7334 AREA:0x0812a060
t:0x4f86afa7 CPU:00 INT_HANDLER_EXIT:0x00000000 (0) SIGEVENT:NONE
t:0x4f86b304 CPU:00 INT_HANDLER_ENTR:0x00000000 (0)       PID:1 IP:0xf0056570 AREA:0x00000000
t:0x4f86ca12 CPU:00 INT_HANDLER_EXIT:0x00000000 (0) SIGEVENT:NONE
t:0x4f86cff6 CPU:00 INT_EXIT:0x00000000 (0) inkernel:0x00000f01
t:0x4f86e276 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000000 msg:"" (0x00040116)
t:0x4f86e756 CPU:00 COMM    :SND_MESSAGE   rcvid:0x0000004f pid:159762
t:0x4f86f84a CPU:00 THREAD  :THREPLY       pid:1511472 tid:1
t:0x4f8705dd CPU:00 THREAD  :THREADY       pid:159762 tid:1
t:0x4f8707d4 CPU:00 THREAD  :THRUNNING     pid:159762 tid:1
t:0x4f870bff CPU:00 COMM    :REC_MESSAGE   rcvid:0x0000004f pid:159762
t:0x4f878b6c CPU:00 KER_CALL:MSG_REPLYV/15 rcvid:0x0000004f status:0x00000000
t:0x4f878f4b CPU:00 COMM    :REPLY_MESSAGE tid:1 pid:1511472
t:0x4f8798d2 CPU:00 THREAD  :THREADY       pid:1511472 tid:1
```

The `SND_PULSE_EXE` event indicates that a SIGEV_PULSE was sent to the server connection ID 0x40000002 of `procnto-smp-instr`, but what is it, and who sent it? Thread 12 of the kernel receives it, and then surprisingly creates a new thread 1 in our process (ID 1511472), and starts chatting with it. What we're seeing here is the teardown of our process. It delivers a death pulse to the kernel, and then one of the kernel's threads receives the pulse and creates a thread in the process to clean up.

In the midst of this teardown, an interrupt occurs, its handler runs, and a message is sent to the process with ID 159762. By looking at the initial system information, we can determine that process ID 159762 is `devc-pty`.

Farther down in the trace is the actual death of our **all_classes** process:

```
t:0x4f8faa68 CPU:00 THREAD  :THRUNNING     pid:1 tid:20
t:0x4f8fb09f CPU:00 COMM    :REC_PULSE     scoid:0x40000002 pid:1
t:0x4f8ff1a5 CPU:00 PROCESS :PROCDESTROY   ppid:426022 pid:1511472
```

As you can tell from a very short look at this trace, wading through a trace can be time-consuming, but can give you a great understanding of what exactly is happening in your system.

You can simplify your task by terminating any processes that you don't want to include in the trace, or by filtering the trace data.

Building your own parser

If you want to create your own parser, consider the structure of `traceprinter` as a starting point. This utility consists of a long list of callback definitions, followed by a fairly simple parsing procedure. Each of the callback definitions is for printing.

The following sections give a brief introduction to the building blocks to the parser, and some of the issues you'll need to handle.

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

You typically use the **traceparser** functions as follows:

1. Initialize the traceparser library by calling *traceparser_init()*. You can also use this function to get the state of your parser.
2. Set the traceparser debug mode and specify a `FILE` stream for the debugging output by calling *traceparser_debug()*.
3. Set up callbacks for processing the trace events that you're interested in:

   traceparser_cs()

   Attach a callback to an event

   traceparser_cs_range()

   Attach a callback to a range of events

   When you set up a callback with either of these functions, you can provide a pointer to arbitrary user data to be passed to the callback.
4. Start parsing your trace data by calling *traceparser()*
5. Destroy your parser by calling *traceparser_destroy()*

You can get information about your parser at any time by calling *traceparser_get_info()*.

For more information about these functions, see their entries in the QNX Neutrino *C Library Reference*.

Simple and combine events, event buffer slots, and the `traceevent_t` structure

A *simple event* is an event that can be described in a single event buffer slot; a *combine event* is an event that's 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 a `traceevent_t` structure:

```
typedef _Uint32t __traceentry;

typedef struct traceevent {
  __traceentry header;   /* CPU, event, format */
  __traceentry data[3]; /* event data          */
} traceevent_t;
```

The `traceevent_t` structure is partly opaque—although some details are provided, you shouldn't access the structure 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. This “thin” protocol doesn't burden the instrumented kernel and keeps the `traceevent_t` structure small. The trade-off is that it may take many `traceevent_t` structures to represent a single kernel event.

In order to distinguish simple 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.

The members of the `traceevent_t` structure are as follows:

header

An encoded header that identifies the event, event class, the CPU, and whether this structure represents a simple or combine event. Use the macros described below to extract these pieces.

data

The data associated with the event. The first element of this array holds the least significant bits of the time stamp. The contents of the remaining elements depend on the event; see the “*Current Trace Events and Data*” appendix.

The following macros extract the information from the event header:

_NTO_TRACE_GETCPU(h)

Get the number of the CPU that the event occurred on.

_NTO_TRACE_GETEVENT(h)

Get the type of event.

_NTO_TRACE_GETEVENT_C(h)

Get the class.

_TRACE_GET_STRUCT(h)

Get the flag that indicates whether this is a simple or combine event:

If the flag equals:	The structure is:
_TRACE_STRUCT_CB	The beginning of a combine event
_TRACE_STRUCT_CC	A continuation of a combine event
_TRACE_STRUCT_CE	The end of a combine event
_TRACE_STRUCT_S	A simple event

In order to save space, the header doesn't use the class and event numbers that we saw in "*Class and events*" in the "Events and the Kernel" chapter. Instead the header uses a compact internal representation. For more information on these representations compare, see *_NTO_TRACE_GET*()*, *_NTO_TRACE_SET*()* in the *C Library Reference*.

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.

Timestamps

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 _NTO_TRACE_CONTROLTIME 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 can 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 seconds—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.

Chapter 7
Tutorials

This chapter leads you through some tutorials to help you learn how to use *TraceEvent()* to control event tracing.

These tutorials all follow the same general procedure:

1. Compile the specified C program into a file of the same name, without the **.c** extension.
2. Run the specified `tracelogger` command. Because we're running `tracelogger` in daemon mode, it doesn't start logging events until our program tells it to. This means that you don't have to rush to start your C program; the tracing waits for you.

   The default number of buffers is 32, which produces a rather large file, so 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.
3. In a *separate* terminal window, run the compiled C program. Some examples use options.
4. Watch the first terminal window; a few seconds after you start your C program, `tracelogger` will finish running.
5. If you run the program, it generates its own sample result file. The "tracebuffer" files are binary files that can be interpreted only with the **libtraceparser** library, which the `traceprinter` utility uses.

   If you don't want to 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 license agreement.

The `instrex.h` header file

To reduce repetition and keep the programs simple, we've put some functionality into a header file, **instrex.h**:

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */

/*
 *  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 state 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
```

You'll have to save **instrex.h** in the same directory as the C code in order to compile the programs.

Gathering all events from all classes

In our first example, we'll set up daemon mode to gather *all* events from *all* classes. Here's the source, **all_classes.c**:

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */

#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 -d1

      with iter_number = your choice of 1 through 10

      After you start the example, tracelogger
      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 sample 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.
     * You can specify the number of iterations (i.e., the
     * number of kernel buffers logged) when you start tracelogger.
     */
    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);
}
```

Compile it, and then run `tracelogger` in one window:

```
tracelogger -d1 -n 3 -f all_classes.kev
```

and run the compiled program in another:

```
./all_classes
```

Despite how quickly the program ran, the amount of data it generated is rather overwhelming.

The trace data is in **all_classes.kev**; to examine it, type:

```
traceprinter -f all_classes.kev | less
```

The output from `traceprinter` will look something like this:

```
TRACEPRINTER version 1.02
TRACEPARSER LIBRARY version 1.02
 -- HEADER FILE INFORMATION --
       TRACE_FILE_NAME:: all_classes.kev
            TRACE_DATE:: Wed Jun 24 10:52:58 2009
       TRACE_VER_MAJOR:: 1
       TRACE_VER_MINOR:: 01
   TRACE_LITTLE_ENDIAN:: TRUE
        TRACE_ENCODING:: 16 byte events
       TRACE_BOOT_DATE:: Tue Jun 23 11:47:46 2009
  TRACE_CYCLES_PER_SEC:: 736629000
         TRACE_CPU_NUM:: 1
         TRACE_SYSNAME:: QNX
        TRACE_NODENAME:: localhost
     TRACE_SYS_RELEASE:: 6.4.1
     TRACE_SYS_VERSION:: 2009/05/20-17:35:56EDT
         TRACE_MACHINE:: x86pc
     TRACE_SYSPAGE_LEN:: 2264
TRACE_TRACELOGGER_ARGS:: tracelogger -d1 -n 3 -f all_classes.kev
 -- KERNEL EVENTS --
t:0x4f81e320 CPU:00 CONTROL: BUFFER sequence = 33, num_events = 714
t:0x4f81e320 CPU:00 CONTROL :TIME msb:0x000037b0 lsb(offset):0x4f81e014
t:0x4f82017a CPU:00 PROCESS :PROCCREATE_NAME
                      ppid:0
```

```
                      pid:1
                     name:proc/boot/procnto-smp-instr

...

t:0x4f854048 CPU:00 THREAD  :THCREATE      pid:1511472 tid:1
t:0x4f854140 CPU:00 THREAD  :THRUNNING     pid:1511472 tid:1
t:0x4f854910 CPU:00 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000 empty:0x00000000
t:0x4f856aac CPU:00 KER_CALL:THREAD_DESTROY/47 tid:-1 status_p:0
t:0x4f857dca CPU:00 KER_EXIT:THREAD_DESTROY/47 ret_val:0x00000030 empty:0x00000000
t:0x4f8588d3 CPU:00 KER_CALL:THREAD_DESTROYALL/48 empty:0x00000000 empty:0x00000000
t:0x4f858ed7 CPU:00 THREAD  :THDESTROY     pid:1511472 tid:1
t:0x4f8598b9 CPU:00 THREAD  :THDEAD        pid:1511472 tid:1
t:0x4f859c4c CPU:00 THREAD  :THRUNNING     pid:1 tid:1
t:0x4f85c6e3 CPU:00 COMM    :SND_PULSE_EXE scoid:0x40000002 pid:1
t:0x4f85cecd CPU:00 THREAD  :THRUNNING     pid:1 tid:12
t:0x4f85d5ad CPU:00 THREAD  :THREADY       pid:1 tid:1
t:0x4f85e5b3 CPU:00 COMM    :REC_PULSE     scoid:0x40000002 pid:1
t:0x4f860ee2 CPU:00 KER_CALL:THREAD_CREATE/46 func_p:f0023170 arg_p:eff6e000
t:0x4f8624c7 CPU:00 THREAD  :THCREATE      pid:1511472 tid:1
t:0x4f8625ff CPU:00 THREAD  :THWAITTHREAD  pid:1 tid:12
t:0x4f8627b4 CPU:00 THREAD  :THRUNNING     pid:1511472 tid:1
t:0x4f8636fd CPU:00 THREAD  :THREADY       pid:1 tid:12
t:0x4f865c34 CPU:00 KER_CALL:CONNECT_SERVER_INFO/41 pid:0 coid:0x00000000
t:0x4f866836 CPU:00 KER_EXIT:CONNECT_SERVER_INFO/41 coid:0x00000000 info->nd:0
t:0x4f86735e CPU:00 KER_CALL:TIMER_TIMEOUT/75 timeout_flags:0x00000050 ntime(sec):30
t:0x4f868445 CPU:00 KER_EXIT:TIMER_TIMEOUT/75 prev_timeout_flags:0x00000000 otime(sec):0
t:0x4f8697d3 CPU:00 INT_ENTR:0x00000000 (0)       IP:0xf008433e
t:0x4f86a276 CPU:00 INT_HANDLER_ENTR:0x00000000 (0)       PID:126997 IP:0x080b7334 AREA:0x0812a060
t:0x4f86afa7 CPU:00 INT_HANDLER_EXIT:0x00000000 (0) SIGEVENT:NONE
t:0x4f86b304 CPU:00 INT_HANDLER_ENTR:0x00000000 (0)       PID:1 IP:0xf0056570 AREA:0x00000000
t:0x4f86ca12 CPU:00 INT_HANDLER_EXIT:0x00000000 (0) SIGEVENT:NONE
t:0x4f86cff6 CPU:00 INT_EXIT:0x00000000 (0) inkernel:0x00000f01
t:0x4f86e276 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000000 msg:"" (0x00040116)
t:0x4f86e756 CPU:00 COMM    :SND_MESSAGE   rcvid:0x0000004f pid:159762
t:0x4f86f84a CPU:00 THREAD  :THREPLY       pid:1511472 tid:1
t:0x4f8705dd CPU:00 THREAD  :THREADY       pid:159762 tid:1
t:0x4f8707d4 CPU:00 THREAD  :THRUNNING     pid:159762 tid:1
t:0x4f870bff CPU:00 COMM    :REC_MESSAGE   rcvid:0x0000004f pid:159762
t:0x4f878b6c CPU:00 KER_CALL:MSG_REPLYV/15 rcvid:0x0000004f status:0x00000000
t:0x4f878f4b CPU:00 COMM    :REPLY_MESSAGE tid:1 pid:1511472
t:0x4f8798d2 CPU:00 THREAD  :THREADY       pid:1511472 tid:1
t:0x4f879db8 CPU:00 KER_EXIT:MSG_REPLYV/15 ret_val:0 empty:0x00000000
t:0x4f87a84f CPU:00 KER_CALL:MSG_RECEIVEV/14 chid:0x00000001 rparts:1

...
```

This example demonstrates the capability of the trace module to capture *huge* amounts of data about the events. The first part of the trace data is the initial state information about all the running processes; to suppress it, start the tracing with _NTO_TRACE_STARTNOSTATE instead of _NTO_TRACE_START.

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

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 "*Classes and events*" in the Events and the Kernel chapter in this guide.

Here's the source, **one_class.c**:

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */

#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 -d1

      with iter_number = your choice of 1 through 10

      After you start the example, tracelogger
      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 sample 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.
     * You can specify the number of iterations (i.e., the
     * number of kernel buffers logged) when you start tracelogger.
     */
    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);
}
```

Compile it, and then run `tracelogger` in one window:

```
tracelogger -d1 -n 3 -f one_class.kev
```

and run the compiled program in another:

```
./one_class
```

The trace data is in **one_class.kev**; to examine it, type:

```
traceprinter -f one_class.kev | less
```

The output from `traceprinter` will look something like this:

```
TRACEPRINTER version 1.02
TRACEPARSER LIBRARY version 1.02
 -- HEADER FILE INFORMATION --
       TRACE_FILE_NAME:: one_class.kev
            TRACE_DATE:: Wed Jun 24 10:55:05 2009
       TRACE_VER_MAJOR:: 1
       TRACE_VER_MINOR:: 01
   TRACE_LITTLE_ENDIAN:: TRUE
        TRACE_ENCODING:: 16 byte events
       TRACE_BOOT_DATE:: Tue Jun 23 11:47:46 2009
  TRACE_CYCLES_PER_SEC:: 736629000
         TRACE_CPU_NUM:: 1
         TRACE_SYSNAME:: QNX
        TRACE_NODENAME:: localhost
     TRACE_SYS_RELEASE:: 6.4.1
     TRACE_SYS_VERSION:: 2009/05/20-17:35:56EDT
         TRACE_MACHINE:: x86pc
     TRACE_SYSPAGE_LEN:: 2264
TRACE_TRACELOGGER_ARGS:: tracelogger -d1 -n 3 -f one_class.kev
 -- KERNEL EVENTS --
t:0x002c4d55 CPU:00 CONTROL: BUFFER sequence = 37, num_events = 714
t:0x002c4d55 CPU:00 THREAD  :THCREATE      pid:1 tid:1
t:0x002c5531 CPU:00 THREAD  :THREADY       pid:1 tid:1 priority:0 policy:1
t:0x002c5bbe CPU:00 THREAD  :THCREATE      pid:1 tid:2
t:0x002c5cd2 CPU:00 THREAD  :THRECEIVE     pid:1 tid:2 priority:255 policy:2
t:0x002c6185 CPU:00 THREAD  :THCREATE      pid:1 tid:3
t:0x002c6272 CPU:00 THREAD  :THRECEIVE     pid:1 tid:3 priority:255 policy:2
t:0x002c64eb CPU:00 THREAD  :THCREATE      pid:1 tid:4
t:0x002c65d8 CPU:00 THREAD  :THRECEIVE     pid:1 tid:4 priority:10 policy:2
t:0x002c67fc CPU:00 THREAD  :THCREATE      pid:1 tid:5
```

```
t:0x002c68ea CPU:00 THREAD  :THRECEIVE     pid:1 tid:5 priority:255 policy:2
t:0x002c6bae CPU:00 THREAD  :THCREATE      pid:1 tid:7
t:0x002c6c9b CPU:00 THREAD  :THRECEIVE     pid:1 tid:7 priority:10 policy:2
t:0x002c6f03 CPU:00 THREAD  :THCREATE      pid:1 tid:8
t:0x002c6ff0 CPU:00 THREAD  :THRECEIVE     pid:1 tid:8 priority:10 policy:2
t:0x002c72ec CPU:00 THREAD  :THCREATE      pid:1 tid:10
t:0x002c73d9 CPU:00 THREAD  :THRECEIVE     pid:1 tid:10 priority:10 policy:2
t:0x002c7665 CPU:00 THREAD  :THCREATE      pid:1 tid:11
t:0x002c7752 CPU:00 THREAD  :THRECEIVE     pid:1 tid:11 priority:10 policy:2
t:0x002c7a9d CPU:00 THREAD  :THCREATE      pid:1 tid:12
t:0x002c7b8a CPU:00 THREAD  :THRECEIVE     pid:1 tid:12 priority:10 policy:2
t:0x002c7e44 CPU:00 THREAD  :THCREATE      pid:1 tid:15
t:0x002c7f31 CPU:00 THREAD  :THRECEIVE     pid:1 tid:15 priority:10 policy:2
t:0x002c81a2 CPU:00 THREAD  :THCREATE      pid:1 tid:20
t:0x002c828f CPU:00 THREAD  :THRECEIVE     pid:1 tid:20 priority:10 policy:2
t:0x002c88e3 CPU:00 THREAD  :THCREATE      pid:2 tid:1
t:0x002c89d3 CPU:00 THREAD  :THREPLY       pid:2 tid:1 priority:10 policy:3
t:0x002c8fad CPU:00 THREAD  :THCREATE      pid:4099 tid:1
t:0x002c909a CPU:00 THREAD  :THRECEIVE     pid:4099 tid:1 priority:10 policy:3
t:0x002c95b7 CPU:00 THREAD  :THCREATE      pid:4100 tid:1
t:0x002c96a4 CPU:00 THREAD  :THRECEIVE     pid:4100 tid:1 priority:10 policy:3
t:0x002c9b6e CPU:00 THREAD  :THCREATE      pid:4101 tid:1
t:0x002c9ccd CPU:00 THREAD  :THSIGWAITINFO pid:4101 tid:1 priority:10 policy:3

...
```

Notice that we've significantly reduced the amount of output.

Gathering five events from four classes

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.

Here's the source, **five_events.c**:

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */

#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 -d1

      with iter_number = your choice of 1 through 10

      After you start the example, tracelogger
      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_KERCALLEXIT,
     * event __KER_MSG_READV.
     */
    TRACE_EVENT
    (
     argv[0],
     TraceEvent(_NTO_TRACE_ADDEVENT, _NTO_TRACE_KERCALLEXIT, __KER_MSG_READV)
    );

    /*
     * Start tracing process
     *
     * During the tracing process, the tracelogger (which
     * is being executed in a daemon mode) will log all events.
     * You can specify the number of iterations (i.e., the
     * number of kernel buffers logged) when you start tracelogger.
     */
    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);
}
```

Compile it, and then run `tracelogger` in one window:

```
tracelogger -d1 -n 3 -f five_events.kev
```


and run the compiled program in another:

```
./five_events
```

The trace data is in **five_events.kev**; to examine it, type:

```
traceprinter -f five_events.kev | less
```

The output from `traceprinter` will look something like this:

```
TRACEPRINTER version 1.02
TRACEPARSER LIBRARY version 1.02
 -- HEADER FILE INFORMATION --
       TRACE_FILE_NAME:: five_events.kev
            TRACE_DATE:: Wed Jun 24 10:56:04 2009
       TRACE_VER_MAJOR:: 1
       TRACE_VER_MINOR:: 01
   TRACE_LITTLE_ENDIAN:: TRUE
        TRACE_ENCODING:: 16 byte events
       TRACE_BOOT_DATE:: Tue Jun 23 11:47:46 2009
  TRACE_CYCLES_PER_SEC:: 736629000
         TRACE_CPU_NUM:: 1
         TRACE_SYSNAME:: QNX
        TRACE_NODENAME:: localhost
     TRACE_SYS_RELEASE:: 6.4.1
     TRACE_SYS_VERSION:: 2009/05/20-17:35:56EDT
         TRACE_MACHINE:: x86pc
     TRACE_SYSPAGE_LEN:: 2264
TRACE_TRACELOGGER_ARGS:: tracelogger -d1 -n 3 -f five_events.kev
 -- KERNEL EVENTS --
t:0x1a14da34 CPU:00 CONTROL: BUFFER sequence = 41, num_events = 714
t:0x1a14da34 CPU:00 PROCESS :PROCCREATE_NAME
                      ppid:0
                       pid:1
                      name:proc/boot/procnto-smp-instr
t:0x1a14ea7d CPU:00 THREAD  :THCREATE      pid:1 tid:1
t:0x1a14ed04 CPU:00 THREAD  :THCREATE      pid:1 tid:2

...

t:0x1a1cc345 CPU:00 THREAD  :THRUNNING     pid:1 tid:15 priority:10 policy:2
t:0x1a1d01b9 CPU:00 THREAD  :THRUNNING     pid:8200 tid:5 priority:10 policy:3
t:0x1a1d6424 CPU:00 INT_ENTR:0x00000000 (0)       IP:0xb8229890
t:0x1a1ed261 CPU:00 THREAD  :THRUNNING     pid:1 tid:4 priority:10 policy:2
t:0x1a1f0016 CPU:00 THREAD  :THRUNNING     pid:426022 tid:1 priority:10 policy:2

...

t:0x2e77ebc5 CPU:00 THREAD  :THRUNNING     pid:1613871 tid:1 priority:10 policy:2
t:0x2e78598d CPU:00 THREAD  :THRUNNING     pid:8200 tid:5 priority:10 policy:3
t:0x2e7ac4fc CPU:00 INT_ENTR:0x00000000 (0)       IP:0xb8229f61
t:0x2e7cec3b CPU:00 KER_EXIT:MSG_READV/16
                    rbytes:22540
                      rmsg:"" (0x1a15080f 0x6e696273 0x6e69742f)
t:0x2e7da478 CPU:00 THREAD  :THRUNNING     pid:1003562 tid:1 priority:10 policy:2
t:0x2e7dc288 CPU:00 THREAD  :THRUNNING     pid:1 tid:15 priority:10 policy:2

...
```

We've now begun to selectively pick and choose events—the massive amount of data is now much more manageable.

Gathering kernel calls

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.

Here's the source, **ker_calls.c**:

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */

#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 -d1

      with iter_number = your choice of 1 through 10

      After you start the example, tracelogger
      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 sample 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_KERCALL, call_num)
    );
} else {
    TRACE_EVENT
    (
     argv[0],
     TraceEvent(_NTO_TRACE_ADDCLASS, _NTO_TRACE_KERCALL)
    );
}

/*
 * Start tracing process
 *
 * During the tracing process, the tracelogger (which
 * is being executed in a daemon mode) will log all events.
 * You can specify the number of iterations (i.e., the
 * number of kernel buffers logged) when you start tracelogger.
 */
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);
}
```

Compile it, and then run `tracelogger` in one window:

```
tracelogger -d1 -n 3 -f ker_calls.all.kev
```

and run the compiled program in another:

```
./ker_calls
```

The trace data is in **ker_calls.all.kev**; to examine it, type:

```
traceprinter -f ker_calls.all.kev | less
```

The output from `traceprinter` will look something like this:

```
TRACEPRINTER version 1.02
TRACEPARSER LIBRARY version 1.02
 -- HEADER FILE INFORMATION --
       TRACE_FILE_NAME:: ker_calls.all.kev
            TRACE_DATE:: Wed Jun 24 10:57:01 2009
       TRACE_VER_MAJOR:: 1
       TRACE_VER_MINOR:: 01
   TRACE_LITTLE_ENDIAN:: TRUE
        TRACE_ENCODING:: 16 byte events
       TRACE_BOOT_DATE:: Tue Jun 23 11:47:46 2009
  TRACE_CYCLES_PER_SEC:: 736629000
         TRACE_CPU_NUM:: 1
         TRACE_SYSNAME:: QNX
        TRACE_NODENAME:: localhost
     TRACE_SYS_RELEASE:: 6.4.1
     TRACE_SYS_VERSION:: 2009/05/20-17:35:56EDT
         TRACE_MACHINE:: x86pc
     TRACE_SYSPAGE_LEN:: 2264
TRACE_TRACELOGGER_ARGS:: tracelogger -d1 -n 3 -f ker_calls.all.kev
 -- KERNEL EVENTS --
t:0x463ad541 CPU:00 CONTROL: BUFFER sequence = 45, num_events = 714
t:0x463ad541 CPU:00 THREAD  :THCREATE      pid:1 tid:1
t:0x463adbe1 CPU:00 THREAD  :THREADY       pid:1 tid:1 priority:0 policy:1
t:0x463adfa8 CPU:00 THREAD  :THCREATE      pid:1 tid:2
t:0x463ae098 CPU:00 THREAD  :THRECEIVE     pid:1 tid:2 priority:255 policy:2
t:0x463ae375 CPU:00 THREAD  :THCREATE      pid:1 tid:3

...

t:0x463d59b6 CPU:00 THREAD  :THSIGWAITINFO pid:1658927 tid:2 priority:10 policy:2
t:0x463d5cb2 CPU:00 THREAD  :THCREATE      pid:1663024 tid:1
t:0x463d5da7 CPU:00 THREAD  :THRUNNING     pid:1663024 tid:1 priority:10 policy:2
t:0x463d666e CPU:00 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000 empty:0x00000000
t:0x463d8e65 CPU:00 KER_CALL:THREAD_DESTROY/47
                       tid:-1
                  priority:-1
                  status_p:0
t:0x463da615 CPU:00 KER_EXIT:THREAD_DESTROY/47 ret_val:0x00000030 empty:0x00000000
t:0x463daf0a CPU:00 KER_CALL:THREAD_DESTROYALL/48 empty:0x00000000 empty:0x00000000
t:0x463db531 CPU:00 THREAD  :THDESTROY     pid:1663024 tid:1
t:0x463dc114 CPU:00 THREAD  :THDEAD        pid:1663024 tid:1 priority:10 policy:2
t:0x463dc546 CPU:00 THREAD  :THRUNNING     pid:1 tid:1 priority:0 policy:1
t:0x463df45d CPU:00 THREAD  :THRUNNING     pid:1 tid:4 priority:10 policy:2
t:0x463dfa7f CPU:00 THREAD  :THREADY       pid:1 tid:1 priority:0 policy:1
t:0x463e36b4 CPU:00 KER_CALL:THREAD_CREATE/46
                       pid:1663024
                    func_p:f0023170
```



```
                     arg_p:eff4e000
                     flags:0x00000000
                 stacksize:10116
               stackaddr_p:eff4e264
                exitfunc_p:0
                    policy:0
            sched_priority:0
         sched_curpriority:0
   param.__ss_low_priority:0
       param.__ss_max_repl:0
param.__ss_repl_period.tv_sec:0
param.__ss_repl_period.tv_nsec:0
param.__ss_init_budget.tv_sec:0
param.__ss_init_budget.tv_nsec:0
               param.empty:0
               param.empty:0
                 guardsize:0
                     empty:0
                     empty:0
                     empty:0
t:0x463e50b0 CPU:00 THREAD  :THCREATE      pid:1663024 tid:1
t:0x463e51d0 CPU:00 THREAD  :THWAITTHREAD  pid:1 tid:4 priority:10 policy:2
t:0x463e5488 CPU:00 THREAD  :THRUNNING     pid:1663024 tid:1 priority:10 policy:2
t:0x463e6408 CPU:00 THREAD  :THREADY       pid:1 tid:4 priority:10 policy:2

...
```

The **ker_calls.c** program takes a `-n` option that lets us view only one type of kernel call. Let's run this program again, specifying the number 14, which signifies __KER_MSG_RECEIVE. For a full list of the values associated with the `-n` option, see **/usr/include/sys/kercalls.h**.

Run `tracelogger` in one window:

```
tracelogger -d1 -n 3 -f ker_calls.14.kev
```

and run the program in another:

```
./ker_calls -n 14
```

The trace data is in **ker_calls.14.kev**; to examine it, type:

```
traceprinter -f ker_calls.14.kev | less
```

The output from `traceprinter` will look something like this:

```
TRACEPRINTER version 1.02
TRACEPARSER LIBRARY version 1.02
 -- HEADER FILE INFORMATION --
       TRACE_FILE_NAME:: ker_calls.14.kev
            TRACE_DATE:: Wed Jun 24 13:39:20 2009
       TRACE_VER_MAJOR:: 1
       TRACE_VER_MINOR:: 01
   TRACE_LITTLE_ENDIAN:: TRUE
        TRACE_ENCODING:: 16 byte events
       TRACE_BOOT_DATE:: Tue Jun 23 11:47:46 2009
  TRACE_CYCLES_PER_SEC:: 736629000
         TRACE_CPU_NUM:: 1
         TRACE_SYSNAME:: QNX
        TRACE_NODENAME:: localhost
     TRACE_SYS_RELEASE:: 6.4.1
     TRACE_SYS_VERSION:: 2009/05/20-17:35:56EDT
         TRACE_MACHINE:: x86pc
     TRACE_SYSPAGE_LEN:: 2264
TRACE_TRACELOGGER_ARGS:: tracelogger -d1 -n 3 -f ker_calls.14.kev
 -- KERNEL EVENTS --
t:0x73bf28d0 CPU:00 CONTROL: BUFFER sequence = 62, num_events = 714
t:0x73bf28d0 CPU:00 THREAD  :THCREATE      pid:1 tid:1
t:0x73bf2e16 CPU:00 THREAD  :THREADY       pid:1 tid:1 priority:0 policy:1
t:0x73bf3203 CPU:00 THREAD  :THCREATE      pid:1 tid:2
```

```
...

t:0x73c21746 CPU:00 THREAD  :THRUNNING     pid:1 tid:1 priority:0 policy:1
t:0x73c24352 CPU:00 THREAD  :THRUNNING     pid:1 tid:15 priority:10 policy:2
t:0x73c247e0 CPU:00 THREAD  :THREADY       pid:1 tid:1 priority:0 policy:1
t:0x73c2547b CPU:00 KER_EXIT:MSG_RECEIVEV/14
                     rcvid:0x00000000
                      rmsg:"" (0x00000000 0x00000081 0x001dd030)
                  info->nd:0
               info->srcnd:0
                 info->pid:1953840
                 info->tid:1
                info->chid:1
               info->scoid:1073741874
                info->coid:1073741824
              info->msglen:0
           info->srcmsglen:56
           info->dstmsglen:24
            info->priority:10
               info->flags:0
            info->reserved:0
t:0x73c29270 CPU:00 THREAD  :THCREATE      pid:1953840 tid:1
t:0x73c293ca CPU:00 THREAD  :THWAITTHREAD  pid:1 tid:15 priority:10 policy:2
t:0x73c2964a CPU:00 THREAD  :THRUNNING     pid:1953840 tid:1 priority:10 policy:2
t:0x73c2a36c CPU:00 THREAD  :THREADY       pid:1 tid:15 priority:10 policy:2
t:0x73c2fccc CPU:00 THREAD  :THREPLY       pid:1953840 tid:1 priority:10 policy:2
t:0x73c30f6b CPU:00 THREAD  :THREADY       pid:159762 tid:1 priority:10 policy:3
t:0x73c311b0 CPU:00 THREAD  :THRUNNING     pid:159762 tid:1 priority:10 policy:3
t:0x73c31835 CPU:00 KER_EXIT:MSG_RECEIVEV/14
                     rcvid:0x0000004f
                      rmsg:"" (0x00040116 0x00000000 0x00000004)
                  info->nd:0
               info->srcnd:0
                 info->pid:1953840
                 info->tid:1
                info->chid:1
               info->scoid:1073741903
                info->coid:0
              info->msglen:4
           info->srcmsglen:4
           info->dstmsglen:0
            info->priority:10
               info->flags:0
            info->reserved:0
t:0x73c3a359 CPU:00 THREAD  :THREADY       pid:1953840 tid:1 priority:10 policy:2
t:0x73c3af50 CPU:00 KER_CALL:MSG_RECEIVEV/14 chid:0x00000001 rparts:1
t:0x73c3b25e CPU:00 THREAD  :THRECEIVE     pid:159762 tid:1 priority:10 policy:3

...
```

Event handling - simple

In this example, we intercept two events from two different classes. Each event has an event handler attached to it; the event handlers are closing and opening the stream. Here's the source, **eh_simple.c**:

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */

#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 you start the example, tracelogger
      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 sample 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;
uint32_t     data_array_1[20]; /* 20 elements for potential args. */

event_data_t e_d_2;
uint32_t     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 __KER_MSG_SENDV
 * from the _NTO_TRACE_KERCALL class.
 */
int call_msg_send_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_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
     */
    TRACE_EVENT(argv[0], TraceEvent(_NTO_TRACE_SETALLCLASSESFAST));

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

        return (-1);
    }

    /*
     * Intercept one event from class _NTO_TRACE_KERCALL,
     * event __KER_MSG_SENDV.
     */
    TRACE_EVENT
    (
     argv[0],
```

```
 TraceEvent(_NTO_TRACE_ADDEVENT, _NTO_TRACE_KERCALLENTER, __KER_MSG_SENDV)
);

/*
 * Add event handler to the event __KER_MSG_SENDV
 * from _NTO_TRACE_KERCALL class.
 */
TRACE_EVENT
(
 argv[0],
 TraceEvent(_NTO_TRACE_ADDEVENTHANDLER, _NTO_TRACE_KERCALLENTER,
            __KER_MSG_SENDV, call_msg_send_eh, &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 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_eh, &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 iterations has been specified as 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_KERCALLENTER, __KER_MSG_SENDV)
    );
    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);
}
```

Compile it, and then run `tracelogger` in one window:

```
tracelogger -d1 -n 1 -f eh_simple.kev
```

For this example, we've specified the number of iterations to be 1.

Run the compiled program in another window:

```
./eh_simple
```

The trace data is in **eh_simple.kev**; to examine it, type:

```
traceprinter -f eh_simple.kev | less
```

The output from `traceprinter` will look something like this:

```
TRACEPRINTER version 1.02
TRACEPARSER LIBRARY version 1.02
 -- HEADER FILE INFORMATION --
       TRACE_FILE_NAME:: eh_simple.kev
            TRACE_DATE:: Wed Jun 24 10:58:41 2009
       TRACE_VER_MAJOR:: 1
       TRACE_VER_MINOR:: 01
   TRACE_LITTLE_ENDIAN:: TRUE
        TRACE_ENCODING:: 16 byte events
       TRACE_BOOT_DATE:: Tue Jun 23 11:47:46 2009
  TRACE_CYCLES_PER_SEC:: 736629000
         TRACE_CPU_NUM:: 1
         TRACE_SYSNAME:: QNX
        TRACE_NODENAME:: localhost
     TRACE_SYS_RELEASE:: 6.4.1
     TRACE_SYS_VERSION:: 2009/05/20-17:35:56EDT
         TRACE_MACHINE:: x86pc
     TRACE_SYSPAGE_LEN:: 2264
TRACE_TRACELOGGER_ARGS:: tracelogger -d1 -n 1 -f eh_simple.kev
 -- KERNEL EVENTS --
t:0x33139a74 CPU:00 CONTROL: BUFFER sequence = 53, num_events = 482
t:0x33139a74 CPU:00 THREAD  :THRUNNING     pid:1749040 tid:1
t:0x362d0710 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000003 msg:"" (0x00100102)
t:0x362d1d04 CPU:00 THREAD  :THRUNNING     pid:217117 tid:1
t:0x362e8e3e CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000000 msg:"" (0x00200113)
t:0x362ea264 CPU:00 THREAD  :THRUNNING     pid:4102 tid:8
t:0x362f1248 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000003 msg:"" (0x00000106)
t:0x362f1c67 CPU:00 THREAD  :THRUNNING     pid:217117 tid:1
t:0x362fd08b CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000003 msg:"" (0x00100102)
t:0x362fd949 CPU:00 THREAD  :THRUNNING     pid:217117 tid:1
t:0x36305424 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000003 msg:"" (0x00000106)
t:0x36305e35 CPU:00 THREAD  :THRUNNING     pid:217117 tid:1
t:0x3630a572 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000003 msg:"" (0x00000106)
t:0x3630aeb7 CPU:00 THREAD  :THRUNNING     pid:217117 tid:1
```

```
t:0x3631bd5b CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000003 msg:"" (0x00100102)
t:0x3631c6aa CPU:00 THREAD  :THRUNNING     pid:217117 tid:1
t:0x363253bb CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000003 msg:"" (0x00000106)
t:0x36325d95 CPU:00 THREAD  :THRUNNING     pid:217117 tid:1
t:0x369b2349 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000003 msg:"" (0x00000106)
t:0x369b2bbe CPU:00 THREAD  :THRUNNING     pid:217117 tid:1
t:0x369b88d8 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000007 msg:"" (0x00100106)
t:0x369b974a CPU:00 THREAD  :THRUNNING     pid:1 tid:15
t:0x369c48ab CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000008 msg:"" (0x00100106)
t:0x369c53db CPU:00 THREAD  :THRUNNING     pid:126997 tid:2
t:0x369cee17 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000008 msg:"" (0x00100106)
t:0x369cf533 CPU:00 THREAD  :THRUNNING     pid:126997 tid:2
t:0x369d82b6 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000009 msg:"" (0x00100106)
t:0x369d9178 CPU:00 THREAD  :THRUNNING     pid:8200 tid:10
t:0x369eae84 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000006 msg:"" (0x00020100)
t:0x369eb687 CPU:00 THREAD  :THRUNNING     pid:1 tid:15
t:0x369f56b1 CPU:00 KER_CALL:MSG_SENDV/11 coid:0x00000006 msg:"" (0x00020100)

...
```

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

Inserting a user simple event

This example demonstrates the insertion of a user event into the event stream. Here's the source, **usr_event_simple.c**:

```
/*
 * $QNXLicenseC:
 * Copyright 2007, QNX Software Systems. All Rights Reserved.
 *
 * You must obtain a written license from and pay applicable license fees to QNX
 * Software Systems before you may reproduce, modify or distribute this software,
 * or any work that includes all or part of this software.   Free development
 * licenses are available for evaluation and non-commercial purposes.  For more
 * information visit http://licensing.qnx.com or email licensing@qnx.com.
 *
 * This file may contain contributions from others.  Please review this entire
 * file for other proprietary rights or license notices, as well as the QNX
 * Development Suite License Guide at http://licensing.qnx.com/license-guide/
 * for other information.
 * $
 */

#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 -d1

      with iter_number = your choice of 1 through 10

      After you start the example, tracelogger
      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 IDs
      (EVENT) equal to: 111, 222, 333, 444 and 555.

      See accompanied documentation and comments within
      the sample 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.
     * You can specify the number of iterations (i.e., the
     * number of kernel buffers logged) when you start tracelogger.
     */
    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 IDs: 111, 222, 333, 444, and 555
     * (possible values are in the range 0...1023).
     * The user events with ID=(111, ..., 444) are simple events
     * that have two numbers attached: ({1,11}, ..., {4,44}).
     * The user string event (ID 555) includes the string,
     * "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);
}
```

Compile it, and then run `tracelogger` in one window:

```
tracelogger -d1 -n 3 -f usr_event_simple.kev
```

and run the compiled program in another:

```
./usr_event_simple
```

The trace data is in **usr_event_simple.kev**; to examine it, type:

```
traceprinter -f usr_event_simple.kev | less
```

The output from `traceprinter` will look something like this:

```
TRACEPRINTER version 1.02
TRACEPARSER LIBRARY version 1.02
 -- HEADER FILE INFORMATION --
       TRACE_FILE_NAME:: usr_event_simple.kev
            TRACE_DATE:: Wed Jun 24 10:59:34 2009
       TRACE_VER_MAJOR:: 1
       TRACE_VER_MINOR:: 01
   TRACE_LITTLE_ENDIAN:: TRUE
        TRACE_ENCODING:: 16 byte events
```

```
       TRACE_BOOT_DATE:: Tue Jun 23 11:47:46 2009
  TRACE_CYCLES_PER_SEC:: 736629000
         TRACE_CPU_NUM:: 1
         TRACE_SYSNAME:: QNX
        TRACE_NODENAME:: localhost
     TRACE_SYS_RELEASE:: 6.4.1
     TRACE_SYS_VERSION:: 2009/05/20-17:35:56EDT
         TRACE_MACHINE:: x86pc
     TRACE_SYSPAGE_LEN:: 2264
TRACE_TRACELOGGER_ARGS:: tracelogger -d1 -n 3 -f usr_event_simple.kev
 -- KERNEL EVENTS --
t:0x254620e4 CPU:00 CONTROL: BUFFER sequence = 54, num_events = 714

...

t:0x25496c81 CPU:00 PROCESS :PROCCREATE_NAME
                      ppid:426022
                       pid:1810480
                      name:./usr_event_simple
t:0x2549701a CPU:00 THREAD  :THCREATE      pid:1810480 tid:1
t:0x25497112 CPU:00 THREAD  :THRUNNING     pid:1810480 tid:1
t:0x2549793a CPU:00 KER_EXIT:TRACE_EVENT/01 ret_val:0x00000000 empty:0x00000000
t:0x25497f48 CPU:00 USREVENT:EVENT:111, d0:0x00000001 d1:0x0000000b
t:0x254982c5 CPU:00 USREVENT:EVENT:222, d0:0x00000002 d1:0x00000016
t:0x25498638 CPU:00 USREVENT:EVENT:333, d0:0x00000003 d1:0x00000021
t:0x25498996 CPU:00 USREVENT:EVENT:444, d0:0x00000004 d1:0x0000002c
t:0x25499451 CPU:00 USREVENT:EVENT:555 STR:"Hello world"
t:0x2549bde5 CPU:00 KER_CALL:THREAD_DESTROY/47 tid:-1 status_p:0
t:0x2549d0d6 CPU:00 KER_EXIT:THREAD_DESTROY/47 ret_val:0x00000030 empty:0x00000000
t:0x2549d8ae CPU:00 KER_CALL:THREAD_DESTROYALL/48 empty:0x00000000 empty:0x00000000

...
```

Inserting your own events lets you 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.

Appendix A
Sample programs

This appendix includes some sample data-capture and parsing programs:

- a simple program that captures trace data
- a companion program that parses event data

These programs could be a useful starting point if you want to write your own event data collection program, especially if you want to do any on-the-fly data collection/parsing for system robustness control.

Data-capture program

This program creates the kernel trace buffers, captures the data, and writes it to standard output. You'd normally redirect the output to a file.

The basic steps in this program are as follows:

1. Get memory for the buffers.
2. Configure the kernel buffers.
3. Indicate which events to collect (thread events and control events).
4. Attach a handler.
5. Start tracing:
   - In the handler, copy the data and schedule a thread.
   - In the thread, write the data to *stdout*.

This program needs to run with the PROCMGR_AID_IO, PROCMGR_AID_MEM_PHYS, and PROCMGR_AID_TRACE abilities enabled.

```
#include <sys/trace.h>
#include <sys/neutrino.h>
#include <sys/mman.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

struct my_buf
{
   unsigned nbytes;
   char data[16*1024];
} my_bufs[4];

int cur_my_buf;

tracebuf_t *kbufs;
paddr_t paddr; // A pointer to physical kernel memory
int hookid;

struct sigevent notify_ev;

const struct sigevent *got_buf( int info )
{
   int ind;
   tracebuf_t *tbuf;
```

```
    ind = _TRACE_GET_BUFFNUM(info);
    tbuf = &kbufs[ind];

    my_bufs[ind].nbytes = tbuf->h.num_events * sizeof(struct traceevent);
    memcpy( my_bufs[ind].data, tbuf->data, my_bufs[ind].nbytes );

    notify_ev.sigev_value.sival_int = ind;
    return &notify_ev;
}

void sig_cleanup(int signo )
{
    TraceEvent( _NTO_TRACE_STOP );
    TraceEvent(_NTO_TRACE_DEALLOCBUFFER);
    InterruptDetach( hookid );
}

int main()
{
    int ret;
    int coid;
    int chid;
    struct _pulse pmsg;
    int rcvid;

    signal( SIGINT, sig_cleanup );

    ret = ThreadCtl(_NTO_TCTL_IO, 0);
    if (-1 == ret )
    {
        perror( "ThreadCtl");
        return 1;
    }

    // This may be dangerous: if anyone else is tracing, this may break them.  But
    // it should allow us to run again if killed without cleaning up.
    TraceEvent(_NTO_TRACE_DEALLOCBUFFER);
    TraceEvent( _NTO_TRACE_STOP );


    // Ask the kernel for four buffers.
    ret = TraceEvent(_NTO_TRACE_ALLOCBUFFER, 4, &paddr);
    if( -1 == ret )
    {
        perror( "TraceEvent Alloc Bufs");
        return 1;
    }
```



```
// Get a vaddr for this memory.
kbufs = mmap_device_memory(0, 4*sizeof(tracebuf_t), PROT_READ|PROT_WRITE, 0, paddr
if( MAP_FAILED == kbufs )
{
   perror("mmap");
   return 1;
}

chid = ChannelCreate(_NTO_CHF_PRIVATE);
coid = ConnectAttach( 0, 0, chid, _NTO_SIDE_CHANNEL, 0 );

SIGEV_PULSE_INIT( &notify_ev, coid, 15, 1, 0 );

// Attach the notification handler for the _NTO_HOOK_TRACE synthetic interrupt
// that the kernel uses to tell us about a full buffer.
hookid = InterruptHookTrace( got_buf, 0 );

// Indicate which events we want.

// Turn off all filters, putting us in default (nothing) state.
TraceEvent(_NTO_TRACE_DELALLCLASSES);
TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_KERCALL);
TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_KERCALL);
TraceEvent(_NTO_TRACE_CLRCLASSPID, _NTO_TRACE_THREAD);
TraceEvent(_NTO_TRACE_CLRCLASSTID, _NTO_TRACE_THREAD);

// Ask for thread, vthread, and process events.
TraceEvent(_NTO_TRACE_ADDCLASS, _NTO_TRACE_THREAD);
TraceEvent(_NTO_TRACE_ADDCLASS, _NTO_TRACE_VTHREAD);
TraceEvent(_NTO_TRACE_ADDCLASS, _NTO_TRACE_PROCESS);

// Ask for control events.
TraceEvent(_NTO_TRACE_ADDCLASS, _NTO_TRACE_CONTROL );

// Set fast mode for all classes.
TraceEvent( _NTO_TRACE_SETALLCLASSESFAST );

// Make sure we are in linear (not ring) mode; as buffers fill,
// we will be notified
TraceEvent(_NTO_TRACE_SETLINEARMODE);

// Start tracing.
TraceEvent( _NTO_TRACE_START );

while(1)
{
   rcvid = MsgReceive( chid, &pmsg, sizeof(pmsg), 0 );
   if( -1 == rcvid )
```

```
        {
           perror("MsgReceive");
           return 1;
        }
        if( 0 == rcvid )
        {
           if (pmsg.code == 1 )
           {
              ret = write( 1, my_bufs[pmsg.value.sival_int].data,
                           my_bufs[pmsg.value.sival_int].nbytes );
              if( -1 == ret )
              {
                 perror("write");
                 return 1;
              }
           }
        }
        // We do not need a message error case, since it is a private channel,
        // so nobody should be able to attach to it or send me messages.
     }
  }
```

Parser

This program reads the events file or output generated by the data-capture program and does some very simple parsing of just the expected events in that data set.

```
#include <sys/trace.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>

// The events for the _NTO_TRACE_THREAD and _NTO_TRACE_VTHREAD classes
// (for the most part) correspond to the thread states defined in <sys/states.h>:
//
//      STATE_DEAD,  /* 0 0x00 */
// STATE_RUNNING,  /* 1 0x01 */
// STATE_READY,  /* 2 0x02 */
// STATE_STOPPED,  /* 3 0x03 */
//
// STATE_SEND,  /* 4 0x04 */
// STATE_RECEIVE,  /* 5 0x05 */
// STATE_REPLY,  /* 6 0x06 */
//
// STATE_STACK,  /* 7 0x07 */
// STATE_WAITTHREAD, /* 8 0x08 */
// STATE_WAITPAGE,  /* 9 0x09 */
//
// STATE_SIGSUSPEND, /* 10 0x0a */
// STATE_SIGWAITINFO, /* 11 0x0b */
// STATE_NANOSLEEP, /* 12 0x0c */
// STATE_MUTEX,  /* 13 0x0d */
// STATE_CONDVAR,  /* 14 0x0e */
// STATE_JOIN,  /* 15 0x0f */
// STATE_INTR,  /* 16 0x10 */
// STATE_SEM,  /* 17 0x11 */
// STATE_WAITCTX,  /* 18 0x12 */
//
// STATE_NET_SEND,  /* 19 0x13 */
// STATE_NET_REPLY, /* 20 0x14 */
//
// STATE_MAX = 24
//
//  There are two additional events:
// enum _TRACE_THREAD_STATE {
// _TRACE_THREAD_CREATE = STATE_MAX,
// _TRACE_THREAD_DESTROY,
// _TRACE_MAX_TH_STATE_NUM
//      };

char *thread_events[26] =
{
 "DEAD",      // 0
 "RUNNING",
 "READY",
 "STOPPED",
 "SEND"   ,
 "RECEIVE", // 5
 "REPLY",
 "STACK",
 "WAITTHREAD",
```

```
 "WAITPAGE",
 "SIGSUSPEND",  //10
 "SIGWAITINFO",
 "NANOSLEEP",
 "MUTEX",
 "CONDVAR",
 "JOIN",   // 15
 "INTR",
 "SEM",
 "WAITCTX",
 "NET_SEND",
 "NET_REPLY", // 20
 "INVAL",
 "INVAL",
 "INVAL", // 23
 "CREATE", // 24
 "DESTROY" // 25
};

void internal_to_external (unsigned int_class, unsigned int_event, unsigned *ext_class, unsigned *ext_event)
{
   int event_64 = 0;

   *ext_class = -1;
   *ext_event = -1;

   switch (int_class)
   {
     case _TRACE_COMM_C:
        *ext_class = _NTO_TRACE_COMM;
        *ext_event = int_event;
        break;

     case _TRACE_CONTROL_C:
        *ext_class = _NTO_TRACE_CONTROL;
        *ext_event = int_event;
        break;

     case _TRACE_INT_C:
        *ext_event = -1;
        switch (int_event)
        {
           case _TRACE_INT_ENTRY:
              *ext_class = _NTO_TRACE_INTENTER;
              break;

           case _TRACE_INT_EXIT:
              *ext_class = _NTO_TRACE_INTEXIT;
              break;

           case _TRACE_INT_HANDLER_ENTRY:
              *ext_class = _NTO_TRACE_INT_HANDLER_ENTER;
              break;

           case _TRACE_INT_HANDLER_EXIT:
              *ext_class = _NTO_TRACE_INT_HANDLER_EXIT;
              break;

           default:
              printf ("Unknown Interrupt event: %d\n", int_event);
        }
```

```
        break;

    case _TRACE_KER_CALL_C:

        /* Remove _NTO_TRACE_KERCALL64 if it's set. */
        if (int_event & _NTO_TRACE_KERCALL64)
        {
            event_64 = 1;
            int_event = int_event & ~_NTO_TRACE_KERCALL64;
        }

        /* Determine the class and event. */
        if (int_event < _TRACE_MAX_KER_CALL_NUM)
        {
            *ext_class = _NTO_TRACE_KERCALLENTER;
            *ext_event = int_event;
        }
        else if (int_event < 2 * _TRACE_MAX_KER_CALL_NUM)
        {
            *ext_class = _NTO_TRACE_KERCALLEXIT;
            *ext_event = int_event - _TRACE_MAX_KER_CALL_NUM;
        }
        else if (int_event < 3 * _TRACE_MAX_KER_CALL_NUM)
        {
            *ext_class = _NTO_TRACE_KERCALLINT;
            *ext_event = int_event - 2 * _TRACE_MAX_KER_CALL_NUM;
        }
        else
        {
            printf ("Unknown kernel event: %d\n", int_event);
        }

        /* Add _NTO_TRACE_KERCALL64 to the external event if it was set for the internal event. */
        if (event_64)
        {
            *ext_event = *ext_event | _NTO_TRACE_KERCALL64;
        }

        break;

    case _TRACE_PR_TH_C:
        *ext_event = -1;
        if (int_event >= (2 * _TRACE_MAX_TH_STATE_NUM))
        {
            *ext_class = _NTO_TRACE_PROCESS;
            *ext_event = 1 << ((int_event >> 6) -1);
        }
        else if (int_event >= _TRACE_MAX_TH_STATE_NUM)
        {
            *ext_class = _NTO_TRACE_VTHREAD;
            *ext_event = 1 << (int_event - _TRACE_MAX_TH_STATE_NUM);
        }
        else
        {
            *ext_class = _NTO_TRACE_THREAD;
            *ext_event = 1 << int_event;
        }
        break;

    case _TRACE_SEC_C:
        *ext_class = _NTO_TRACE_SEC;
```

```
        *ext_event = int_event;
        break;

     case _TRACE_SYSTEM_C:
        *ext_class = _NTO_TRACE_SYSTEM;
        *ext_event = int_event;
        break;

     case _TRACE_USER_C:
        *ext_class = _NTO_TRACE_USER;
        *ext_event = int_event;
        break;

     default:
        printf ("Unknown class: %d\n", int_class);
   }
}

int main()
{
   struct traceevent *tv;
   int i;
   char buf[64 * sizeof *tv ];

   int nb;

   unsigned internal_class, internal_event, cpu;
   unsigned external_class, external_event, event_type;
   char name_buff[1024];
   int name_index;

   while(1)
   {
      nb = read( 0, buf, sizeof(buf));
      if( nb <= 0 )
      {
         return 0;
      }
      for( i = 0; i < nb/sizeof *tv; i++)
      {
         tv = (struct traceevent *)&buf[i*sizeof *tv];

         /* The header includes the internal class and event numbers, the CPU index, and
            the type of event (simple or combine). */
         internal_class = _NTO_TRACE_GETEVENT_C(tv->header);
         internal_event = _NTO_TRACE_GETEVENT(tv->header);
         cpu =  _NTO_TRACE_GETCPU(tv->header);
         event_type = _TRACE_GET_STRUCT(tv->header);
         switch (event_type)
         {
            case _TRACE_STRUCT_S:
               printf("S  ");
               break;
            case _TRACE_STRUCT_CB:
               printf("CB ");
               break;
            case _TRACE_STRUCT_CC:
               printf("CC ");
               break;
            case _TRACE_STRUCT_CE:
               printf("CE ");
```

```
            break;
        default:
            printf("?  ");
    }

    /* Convert the internal class and event numbers into external ones. */
    internal_to_external (internal_class, internal_event, &external_class, &external_event);

    if( _NTO_TRACE_PROCESS == external_class )
    {
        switch (external_event)
        {
            case _NTO_TRACE_PROCCREATE:
                printf("_NTO_TRACE_PROCESS, _NTO_TRACE_PROCCREATE, cpu: %d, timestamp: %8x, ppid:%d, pid:%d\n",
                       cpu, tv->data[0], tv->data[1], tv->data[2]);
                break;
            case _NTO_TRACE_PROCCREATE_NAME:
                if ((event_type == _TRACE_STRUCT_CB) || (event_type == _TRACE_STRUCT_S))
                {
                    /* The first combine event includes the parent's pid and the pid of the new process. */
                    printf("_NTO_TRACE_PROCESS, _NTO_TRACE_PROCCREATE_NAME, cpu: %d, timestamp: %8x, ppid:%d, pid:%d\n",
                           cpu, tv->data[0], tv->data[1], tv->data[2]);
                    memset (name_buff, 0, sizeof(name_buff));
                    name_index = 0;
                }
                else
                {
                    /* Subsequent combine events include parts of the name. */
                    memcpy (name_buff + name_index, &(tv->data[1]), sizeof(unsigned int));
                    memcpy (name_buff + name_index + sizeof(unsigned int), &(tv->data[2]), sizeof(unsigned int));
                    name_index += 2 * sizeof(unsigned int);

                    if (event_type == _TRACE_STRUCT_CE)
                    {
                        /* This is the last of the combine events. */
                        printf("_NTO_TRACE_PROCESS, _NTO_TRACE_PROCCREATE_NAME, cpu: %d, timestamp: %8x, %s\n",
                               cpu, tv->data[0], name_buff);
                    }
                }
                break;
            case _NTO_TRACE_PROCDESTROY:
                printf("_NTO_TRACE_PROCESS, _NTO_TRACE_PROCDESTROY, cpu: %d, timestamp: %8x, ppid:%d, pid:%d\n",
                       cpu, tv->data[0], tv->data[1], tv->data[2]);
                break;
            case _NTO_TRACE_PROCDESTROY_NAME: // Not used.
                printf("_NTO_TRACE_PROCESS, _NTO_TRACE_PROCDESTROY_NAME, cpu: %d, timestamp: %8x, ppid:%d, pid:%d\n",
                       cpu, tv->data[0], tv->data[1], tv->data[2]);
                break;
            case _NTO_TRACE_PROCTHREAD_NAME:
                if ((event_type == _TRACE_STRUCT_CB) || (event_type == _TRACE_STRUCT_S))
                {
                    /* The first combine event includes the pid and tid. */
                    printf("_NTO_TRACE_PROCESS, _NTO_TRACE_PROCTHREAD_NAME, cpu: %d, timestamp: %8x, pid:%d, tid:%d\n",
                           cpu, tv->data[0], tv->data[1], tv->data[2]);
                    memset (name_buff, 0, sizeof(name_buff));
                    name_index = 0;
                }
                else
                {
                    /* Subsequent combine events include parts of the name. */
                    memcpy (name_buff + name_index, &(tv->data[1]), sizeof(unsigned int));
```

```
            memcpy (name_buff + name_index + sizeof(unsigned int), &(tv->data[2]), sizeof(unsigned int));
            name_index += 2 * sizeof(unsigned int);

            if (event_type == _TRACE_STRUCT_CE)
            {
               /* This is the last of the combine events. */
               printf("_NTO_TRACE_PROCESS, _NTO_TRACE_PROCTHREAD_NAME, cpu: %d, timestamp: %8x, %s\n",
                      cpu, tv->data[0], name_buff);
            }
         }
         break;
      default:
         printf("_NTO_TRACE_PROCESS, Unknown event: %d, cpu: %d, timestamp: %8x, pid:%d, tid:%d\n",
                external_event, cpu, tv->data[0], tv->data[1], tv->data[2]);
   }
}
else if (_NTO_TRACE_THREAD == external_class)
{
   /* The data for all the THREAD events includes the process and thread IDs.
      The internal event corresponds to the thread state. */

   printf("_NTO_TRACE_THREAD, _NTO_TRACE_TH%s, cpu: %d, timestamp: %8x, pid:%d, tid:%d\n",
          thread_events[internal_event], cpu, tv->data[0], tv->data[1],
          tv->data[2]);
}
else if (_NTO_TRACE_VTHREAD == external_class)
{
   /* The data for all the VTHREAD events includes the process and thread IDs.
      The internal event corresponds to the thread state. */

   printf("_NTO_TRACE_VTHREAD, _NTO_TRACE_VTH%s, cpu: %d, timestamp: %8x, pid:%d, tid:%d\n",
          thread_events[internal_event], cpu, tv->data[0], tv->data[1],
          tv->data[2]);
}
else if ( _NTO_TRACE_CONTROL == external_class )
{
   switch (external_event)
   {
      case _NTO_TRACE_CONTROLBUFFER:
         /* The data includes the sequence number of the buffer and the number of event slots in it. */
         printf("_NTO_TRACE_CONTROL,  _NTO_TRACE_CONTROLBUFFER, cpu: %d, timestamp: %8x, sequence:%d, number of events:%
                cpu, tv->data[0], tv->data[1], tv->data[2]);
         break;
      case _NTO_TRACE_CONTROLTIME:
         /* The data includes the full time stamp. */
         printf("_NTO_TRACE_CONTROL, _NTO_TRACE_CONTROLTIME, cpu: %d, timestamp: %8x, msb:%x, lsb:%x\n",
                cpu, tv->data[0], tv->data[1], tv->data[2]);
         break;
      default:
         printf("_NTO_TRACE_CONTROL, Unknown event: %d, cpu: %d, timestamp: %8x, d1:%x, d2:%x\n",
                external_event, cpu, tv->data[0], tv->data[1], tv->data[2]);
   }
}
else
{
   printf("Unhandled class: %d, event: %d, cpu: %d, timestamp: %8x, d1:%x, d2:%x\n",
          external_class, external_event, cpu, tv->data[0], tv->data[1], tv->data[2]);
}

}
```

```
    }
}
```

Appendix B
Current Trace Events and Data

This appendix provides a table that lists all the trace events and summarizes the data included for each in both wide and fast modes.

Interpreting the table

As you examine the table, note the following:

- Some of the functions listed below (e.g., *InterruptDetachFunc()*, *SignalFault()*) are internal ones that you won't find documented in the QNX Neutrino *C Library Reference*.
- If a function has a restartable version (with a `_r` in its name), the events for both versions are as listed for the function without the `_r`.
- If a kernel call fails, the exit trace event includes the return code and the error code (e.g., an *errno* value), instead of the data listed below.

As an example, let's look at the events for *MsgSend()*, *MsgSendv()*, and *MsgSendvs()*. As mentioned above, the information is the same for the restartable versions of these functions too.

Here's what the table gives for the entry (_NTO_TRACE_KERCALLENTER) to these functions:

```
Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_SENDV
Fast: coid, msg
Wide: coid, sparts, rparts, msg[0], msg[1], msg[2]
Call: MsgSend,MsgSendv,MsgSendvs
#Args: MSG_SENDV, fHcoid, Dsparts, Drparts, fSmsg, s, s
```

This part describes the __KER_MSG_SENDV trace event that's emitted on entry to the function. In fast mode, the event includes the following data:

Fast mode data	Number of bytes for the event
Connection ID	4 bytes
Message data	4 bytes (the first 4 bytes usually comprise the header)
	Total emitted: 8 bytes

In wide mode, the event includes the following data:

Wide mode data	Number of bytes for the event
Connection ID	4 bytes
# of parts to send	4 bytes
# of parts to receive	4 bytes
Message data	4 bytes (the first 4 bytes usually comprise the header)
Message data	4 bytes
Message data	4 bytes
	Total emitted: 24 bytes

The second (_NTO_TRACE_KERCALLEXIT) part describes the __KER_MSG_SENDV event that's emitted on exit from the function:

```
Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_SENDV
Fast: status, rmsg[0]
Wide: status, rmsg[0], rmsg[1], rmsg[2]
Call: MsgSend,MsgSendv,MsgSendvs
#Args: MSG_SENDV, fDstatus, fSrmsg, s, s
```

In fast mode, the event includes the following data if the kernel call was successful:

Fast mode data	Number of bytes for the event
Exit status	4 bytes
Message data	4 bytes (the first 4 bytes usually comprise the header)
	Total emitted: 8 bytes

In wide mode, the event includes the following data if the kernel call was successful:

Wide mode data	Number of bytes for the event
Exit status	4 bytes
Message data	4 bytes (the first 4 bytes usually comprise the header)
Message data	4 bytes
Message data	4 bytes
	Total emitted: 16 bytes

In both fast and wide mode, the event includes the following data if the kernel call failed:

Fast and wide mode data	Number of bytes for the event
Exit status	4 bytes
Error code	4 bytes
	Total emitted: 8 bytes

For many of the events, you'll see a comment like this:

```
#Args: MSG_SENDV, fHcoid, Dsparts, Drparts, fSmsg, s, s
```

This line indicates how `traceprinter` displays the data associated with the event. The format codes are as follows:

Code	Format
`H`	Hexadecimal (32 bit)
`D`	Decimal (32 bit)
`X`	Hexadecimal (64 bit)
`E`	Decimal (64 bit)
`S`	Begin a character string
`s`	Continue with a character string
`P`	Pointer
`N`	Named string
`f`	Fast mode prefix

For example, `fHcoid` indicates that the connection ID (`coid`) is displayed as a 32-bit hexadecimal number, and it's included in fast mode (and wide mode).

Table of events

```
# The format of this file is as follows
# #'s are comments
# Event blocks are delimited by a blank line
# Class -> The external class description
# Event -> The external event description
# Fast -> Comma delimited list of wide argument subset
# Wide -> Full arguments emitted
# Call -> If the event is associated with a kernel call, put them here

Note - all _NTO_TRACE_KERCALLEXIT calls now have errno as the second
parameter if the kercall failed (i.e., the retval is -1)

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_BAD
Fast: empty, empty
Wide: empty, empty
Call: N/A
#Args: BAD, fHempty, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_BAD
Fast: ret_val, empty
Wide: ret_val, empty
Call: N/A
#Args: BAD, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CACHE_FLUSH
Fast: addr, nlines
Wide: addr, nlines, flags, index
Call: CacheFlush
#Args: CACHE_FLUSH, fHaddr, fHnlines, Hflags, Dindex

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CACHE_FLUSH|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: addr (64), nlines (64), flags, index
Call: CacheFlush
#Args: CACHE_FLUSH, Xaddr, Xnlines, Hflags, Dindex

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CACHE_FLUSH
Fast: ret_val, empty
Wide: ret_val, empty
Call: CacheFlush
#Args: CACHE_FLUSH, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CHANNEL_CREATE
Fast: flags, empty
```

```
Wide: flags, empty
Call: ChannelCreate
#Args: CHANNEL_CREATE, fHflags, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CHANNEL_CREATE
Fast: chid, empty
Wide: chid, empty
Call: ChannelCreate
#Args: CHANNEL_CREATE, fHchid, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CHANNEL_DESTROY
Fast: chid, empty
Wide: chid, empty
Call: ChannelDestroy
#Args: CHANNEL_DESTROY, fHchid, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CHANNEL_DESTROY
Fast: ret_val, empty
Wide: ret_val, empty
Call: ChannelDestroy
#Args: CHANNEL_DESTROY, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CHANCON_ATTR
Fast: chid, flags
Wide: chid, flags, new_attr
Call: ChannelConnectAttr
#Args: CHANCON_ATTR, fHchid, fHflags, ???

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CHANCON_ATTR
Fast: ret_val, empty
Wide: ret_val, old_attr
Call: ChannelConnectAttr
#Args: CHANCON_ATTR, fhret_val, ???

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CLOCK_ADJUST
Fast: id, new->tick_count
Wide: id, new->tick_count, new->tick_nsec_inc
Call: ClockAdjust
#Args: CLOCK_ADJUST, fDid, fDnew->tick_count, Dnew->tick_nsec_inc

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CLOCK_ADJUST
Fast: ret_val, old->tick_count
Wide: ret_val, old->tick_count, old->tick_nsec_inc
Call: ClockAdjust
#Args: CLOCK_ADJUST, fDret_val, fDold->tick_count, Dold->tick_nsec_inc

Class: _NTO_TRACE_KERCALLENTER
```

```
Event: __KER_CLOCK_ID
Fast: pid, tid
Wide: pid, tid
Call: ClockId
#Args: CLOCK_ID, fDpid, fDtid

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CLOCK_ID
Fast: ret_val, empty
Wide: ret_val, empty
Call: ClockId
#Args: CLOCK_ID, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CLOCK_PERIOD
Fast: id, new->nsec
Wide: id, new->nsec, new->fract
Call: ClockPeriod
#Args: CLOCK_PERIOD, fDid, fDnew->nsec, Dnew->fract

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CLOCK_PERIOD
Fast: ret_val, old->nsec
Wide: ret_val, old->nsec, old->fract
Call: ClockPeriod
#Args: CLOCK_PERIOD, fDret_val, fDold->nsec, Dold->fract

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CLOCK_TIME
Fast: id, new(sec)
Wide: id, new(sec), new(nsec)
Call: ClockTime
#Args: CLOCK_TIME, fDid, fDnew(sec), Dnew(nsec)

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CLOCK_TIME
Fast: ret_val, old(sec)
Wide: ret_val, old(sec), old(nsec)
Call: ClockTime
#Args: CLOCK_TIME, fDret_val, fDold(sec), Dold(nsec)

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CONNECT_ATTACH
Fast: nd, pid
Wide: nd, pid, chid, index, flags
Call: ConnectAttach
#Args: CONNECT_ATTACH, fHnd, fDpid, Hchid, Dindex, Hflags

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CONNECT_ATTACH
Fast: coid, empty
Wide: coid, empty
Call: ConnectAttach
#Args: CONNECT_ATTACH, fHcoid, fHempty
```

```
Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CONNECT_CLIENT_INFO
Fast: scoid, ngroups
Wide: scoid, ngroups
Call: ConnectClientInfo
#Args: CONNECT_CLIENT_INFO, fHscoid, fDngroups

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CONNECT_CLIENT_INFO
Fast: ret_val, info->nd
Wide: ret_val, info->nd, info->pid, info->sid, flags, info->ruid, info->euid, info->suid, info->rgid,
info->egid, info->sgid, info->ngroups, info->grouplist[0], info->grouplist[1], info->grouplist[2],
info->grouplist[3], info->grouplist[4], info->grouplist[5], info->grouplist[6], info->grouplist[7]
Call: ConnectClientInfo
#Args: CONNECT_CLIENT_INFO, fHscoid, fDngroups

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CONNECT_DETACH
Fast: coid, empty
Wide: coid, empty
Call: ConnectDetach
#Args: CONNECT_DETACH, fHcoid, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CONNECT_DETACH
Fast: ret_val, empty
Wide: ret_val, empty
Call: ConnectDetach
#Args: CONNECT_DETACH, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CONNECT_FLAGS
Fast: coid, bits
Wide: pid, coid, masks, bits
Call: ConnectFlags
#Args: CONNECT_FLAGS, Dpid, fHcoid, Hmasks, fHbits

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CONNECT_FLAGS
Fast: old_flags, empty
Wide: old_flags, empty
Call: ConnectFlags
#Args: CONNECT_FLAGS, fHold_flags, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_CONNECT_SERVER_INFO
Fast: pid, coid
Wide: pid, coid
Call: ConnectServerInfo
#Args: CONNECT_SERVER_INFO, fDpid, fHcoid

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CONNECT_SERVER_INFO
```

```
Fast: coid, info->nd
Wide: coid, info->nd, info->srcnd, info->pid, info->tid, info->chid, info->scoid, info->coid,
info->msglen, info->srcmsglen, info->dstmsglen, info->priority, info->flags, info->reserved
Call: ConnectServerInfo
#Args: CONNECT_SERVER_INFO, fDpid, fHcoid

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_CONNECT_SERVER_INFO|_NTO_TRACE_KERCALL64
Fast: coid, info->nd
Wide: coid, info->nd, info->srcnd, info->pid, info->tid, info->chid, info->scoid, info->coid,
info->msglen (64), info->srcmsglen (64), info->dstmsglen (64), info->priority, info->flags, info->reserved[0],
info->reserved[1]
Call: ConnectServerInfo
#Args: CONNECT_SERVER_INFO, fDpid, fHcoid

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_INTERRUPT_ATTACH
Fast: intr, flags
Wide: intr, handler_p, area_p, areasize, flags
Call: InterruptAttach
#Args: INTERRUPT_ATTACH, fDintr, Phandler_p, Parea_p, Dareasize, fHflags

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_INTERRUPT_ATTACH|_NTO_TRACE_KERCALL64
Fast: intr, flags
Wide: intr, handler_p (64), area_p (64), areasize, flags
Call: InterruptAttach
#Args: INTERRUPT_ATTACH, fDintr, Qhandler_p, Qarea_p, Dareasize, fHflags

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_INTERRUPT_ATTACH
Fast: int_fun_id, empty
Wide: int_fun_id, empty
Call: InterruptAttach
#Args: INTERRUPT_ATTACH, fHint_fun_id, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_INTERRUPT_DETACH
Fast: id, empty
Wide: id, empty
Call: InterruptDetach
#Args: INTERRUPT_DETACH, fDid, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_INTERRUPT_DETACH
Fast: ret_val, empty
Wide: ret_val, empty
Call: InterruptDetach
#Args: INTERRUPT_DETACH, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_INTERRUPT_DETACH_FUNC
Fast: intr, handler_p
Wide: intr, handler_p
```

```
Call: N/A
#Args: INTERRUPT_DETACH_FUNC, fDintr, fPhandler_p

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_INTERRUPT_DETACH_FUNC|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: intr, handler_p (64)
Call: N/A
#Args: INTERRUPT_DETACH_FUNC, Dintr, Qhandler_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_INTERRUPT_DETACH_FUNC
Fast: ret_val, empty
Wide: ret_val, empty
Call: N/A
#Args: INTERRUPT_DETACH_FUNC, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_INTERRUPT_MASK
Fast: intr, id
Wide: intr, id
Call: InterruptMask
#Args: INTERRUPT_MASK, fDintr, fDid

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_INTERRUPT_MASK
Fast: mask_level, empty
Wide: mask_level, empty
Call: InterruptMask
#Args: INTERRUPT_MASK, fHmask_level, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_INTERRUPT_UNMASK
Fast: intr, id
Wide: intr, id
Call: InterruptUnmask
#Args: INTERRUPT_UNMASK, fDintr, fDid

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_INTERRUPT_UNMASK
Fast: mask_level, empty
Wide: mask_level, empty
Call: InterruptUnmask
#Args: INTERRUPT_UNMASK, fHmask_level, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_INTERRUPT_WAIT
Fast: flags, timeout_tv_sec
Wide: flags, timeout_tv_sec, timeout_tv_nsec
Call: InterruptWait
#Args: INTERRUPT_WAIT, fHflags, fDtimeout_tv_sec, Dtimeout_tv_nsec

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_INTERRUPT_WAIT
```

```
Fast: ret_val, timeout_p
Wide: ret_val, timeout_p
Call: InterruptWait
#Args: INTERRUPT_WAIT, fDret_val, fPtimeout_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_INTERRUPT_WAIT|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: ret_val, timeout_p (64)
Call: InterruptWait
#Args: INTERRUPT_WAIT, Dret_val, Qtimeout_p

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_INTERRUPT_CHARACTERISTIC
Fast: id, type, new
Wide: id, type, new
Call: InterruptCharacteristic
#Args: INTERRUPT_CHARACTERISTIC, fDid, fHtype, fDnew

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_INTERRUPT_CHARACTERISTIC
Fast: ret_val, old
Wide: ret_val, old
Call: InterruptCharacteristic
#Args: INTERRUPT_CHARACTERISTIC, Dret_val, fHold

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_CURRENT
Fast: rcvid, empty
Wide: rcvid, empty
Call: MsgCurrent
#Args: MSG_CURRENT, fHrcvid, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_CURRENT
Fast: empty, empty
Wide: empty, empty
Call: MsgCurrent
#Args: MSG_CURRENT, fHempty, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_PAUSE
Fast: rcvid, cookie
Wide: rcvid, cookie
Call: MsgPause
#Args: MSG_CURRENT, fHrcvid, fHcookie

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_PAUSE
Fast: empty, empty
Wide: empty, empty
Call: MsgPause
#Args: MSG_CURRENT, fHempty, fHempty
```

```
Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_DELIVER_EVENT
Fast: rcvid, event->sigev_notify
Wide: rcvid, event->sigev_notify, event->sigev_notify_function_p, event->sigev_value,
event->sigev_notify_attributes_p
Call: MsgDeliverEvent

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_DELIVER_EVENT|_NTO_TRACE_KERCALL64
Fast: rcvid, event->sigev_notify
Wide: rcvid, event->sigev_notify, event->sigev_notify_function_p (64), event->sigev_value (64),
event->sigev_notify_attributes_p (64)
Call: MsgDeliverEvent

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_DELIVER_EVENT
Fast: ret_val, eventp
Wide: ret_val, eventp
Call: MsgDeliverEvent
#Args: MSG_DELIVER_EVENT, fHret_val, fPeventp

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_DELIVER_EVENT|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: ret_val, eventp (64)
Call: MsgDeliverEvent
#Args: MSG_DELIVER_EVENT, Hret_val, Qeventp

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_ERROR
Fast: rcvid, err
Wide: rcvid, err
Call: MsgError
#Args: MSG_ERROR, fHrcvid, fDerr

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_ERROR
Fast: ret_val, empty
Wide: ret_val, empty
Call: MsgError
#Args: MSG_ERROR, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_INFO
Fast: rcvid, info_p
Wide: rcvid, info_p
Call: MsgInfo
#Args: MSG_INFO, fHrcvid, fPinfo_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_INFO
Fast: ret_val, info->nd
Wide: ret_val, info->nd, info->srcnd, info->pid, info->tid, info->chid, info->scoid, info->coid,
info->msglen, info->srcmsglen, info->dstmsglen, info->priority, info->flags, info->reserved
```



```
Call: MsgInfo

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_INFO|_NTO_TRACE_KERCALL64
Fast: ret_val, info->nd
Wide: ret_val, info->nd, info->srcnd, info->pid, info->tid, info->chid, info->scoid, info->coid,
info->msglen (64), info->srcmsglen (64), info->dstmsglen (64), info->priority, info->flags, info->reserved[0],
info->reserved[1]
Call: MsgInfo

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_KEYDATA
Fast: rcvid, op
Wide: rcvid, op
Call: MsgKeyData
#Args: MSG_KEYDATA, fHrcvid, fHop

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_KEYDATA
Fast: ret_val, newkey
Wide: ret_val, newkey
Call: MsgKeyData
#Args: MSG_KEYDATA, fHret_val, fDnewkey

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_READV
Fast: rcvid, offset
Wide: rcvid, rmsg_p, rparts, offset
Call: MsgRead,MsgReadv
#Args: MSG_READV, fHrcvid, Prmsg_p, Drparts, fHoffset

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_READV|_NTO_TRACE_KERCALL64
Fast: rcvid, empty
Wide: rcvid, rmsg_p (64), rparts (64), offset (64)
Call: MsgRead,MsgReadv
#Args: MSG_READV, fHrcvid, Qrmsg_p, Erparts, Xoffset

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_READV
Fast: rbytes, rmsg[0]
Wide: rbytes, rmsg[0], rmsg[1], rmsg[2]
Call: MsgRead,MsgReadv
#Args: MSG_READV, fDrbytes, fSrmsg, s, s

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_READV|_NTO_TRACE_KERCALL64
Fast: rbytes (64)
Wide: rbytes, rmsg[0], rmsg[1], rmsg[2]
Call: MsgRead,MsgReadv
#Args: MSG_READV, fErbytes, Srmsg, s, s

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_RECEIVEPULSEV
```

```
Fast: chid, rparts
Wide: chid, rparts
Call: MsgReceivePulse,MsgReceivePulsev
#Args: MSG_RECEIVEPULSEV, fHchid, fDrparts

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_RECEIVEPULSEV|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: chid, rparts (64)
Call: MsgReceivePulse,MsgReceivePulsev
#Args: MSG_RECEIVEPULSEV, Hchid, Erparts

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_RECEIVEPULSEV
Fast: rcvid, rmsg[0]
Wide: rcvid, rmsg[0], rmsg[1], rmsg[2], info->nd, info->srcnd, info->pid, info->tid, info->chid, info->scoid,
info->coid, info->msglen, info->srcmsglen, info->dstmsglen, info->priority, info->flags, info->reserved
Call: MsgReceivePulse,MsgReceivePulsev

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_RECEIVEPULSEV|_NTO_TRACE_KERCALL64
Fast: rcvid, rmsg[0]
Wide: rcvid, rmsg[0], rmsg[1], rmsg[2], info->nd, info->srcnd, info->pid, info->tid, info->chid, info->scoid,
info->coid, info->msglen (64), info->srcmsglen (64), info->dstmsglen (64), info->priority, info->flags,
info->reserved[0], info->reserved[1]
Call: MsgReceivePulse,MsgReceivePulsev

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_RECEIVEV
Fast: chid, rparts
Wide: chid, rparts
Call: MsgReceive,MsgReceivev
#Args: MSG_RECEIVEV, fHchid, fDrparts

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_RECEIVEV|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: chid, rparts (64)
Call: MsgReceive,MsgReceivev
#Args: MSG_RECEIVEV, Hchid, Erparts

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_RECEIVEV
Fast: rcvid, rmsg[0]
Wide: rcvid, rmsg[0], rmsg[1], rmsg[2], info->nd, info->srcnd, info->pid, info->tid, info->chid, info->scoid,
info->coid, info->msglen, info->srcmsglen, info->dstmsglen, info->priority, info->flags, info->reserved
Call: MsgReceive,MsgReceivev

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_RECEIVEV|_NTO_TRACE_KERCALL64
Fast: rcvid, rmsg[0]
Wide: rcvid, rmsg[0], rmsg[1], rmsg[2], info->nd, info->srcnd, info->pid, info->tid, info->chid, info->scoid,
info->coid, info->msglen (64), info->srcmsglen (64), info->dstmsglen (64), info->priority, info->flags,
info->reserved[0], info->reserved[1]
```

```
Call: MsgReceive,MsgReceivev

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_REPLYV
Fast: rcvid, status
Wide: rcvid, sparts, status, smsg[0], smsg[1], smsg[2]
Call: MsgReply,MsgReplyv
#Args: MSG_REPLYV, fHrcvid, Dsparts, fHstatus, Ssmsg, s, s

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_REPLYV|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: rcvid, sparts (64), status (64), smsg[0], smsg[1], smsg[2]
Call: MsgReply,MsgReplyv
#Args: MSG_REPLYV, Hrcvid, Esparts, Xstatus, Ssmsg, s, s

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_REPLYV
Fast: ret_val, empty
Wide: ret_val, empty
Call: MsgReply,MsgReplyv
#Args: MSG_REPLYV, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_SEND_PULSE
Fast: coid, code
Wide: coid, priority, code, value
Call: MsgSendPulse
#Args: MSG_SEND_PULSE, fHcoid, Dpriority, fHcode, Hvalue

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_SEND_PULSE
Fast: status, empty
Wide: status, empty
Call: MsgSendPulse
#Args: MSG_SEND_PULSE, fDstatus, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_SEND_PULSEPTR
Fast: coid, code
Wide: coid, priority, code, value
Call: MsgSendPulse
#Args: MSG_SEND_PULSE, fHcoid, Dpriority, fHcode, Hvalue

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_SEND_PULSEPTR|_NTO_TRACE_KERCALL64
Fast: coid, code
Wide: coid, priority, code, value (64)
Call: MsgSendPulse
#Args: MSG_SEND_PULSE, fHcoid, Dpriority, fHcode, Xvalue

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_SEND_PULSEPTR
Fast: status, empty
```

```
Wide: status, empty
Call: MsgSendPulse
#Args: MSG_SEND_PULSE, fDstatus, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_SENDV
Fast: coid, msg
Wide: coid, sparts, rparts, msg[0], msg[1], msg[2]
Call: MsgSend,MsgSendv,MsgSendvs
#Args: MSG_SENDV, fHcoid, Dsparts, Drparts, fSmsg, s, s

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_SENDV|_NTO_TRACE_KERCALL64
Fast: coid, msg
Wide: coid, sparts (64), rparts (64), msg[0], msg[1], msg[2]
Call: MsgSend,MsgSendv,MsgSendvs
#Args: MSG_SENDV, fHcoid, Esparts, Erparts, fSmsg, s, s

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_SENDV
Fast: status, rmsg[0]
Wide: status, rmsg[0], rmsg[1], rmsg[2]
Call: MsgSend,MsgSendv,MsgSendvs
#Args: MSG_SENDV, fDstatus, fSrmsg, s, s

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_SENDV|_NTO_TRACE_KERCALL64
Fast: status (64)
Wide: status (64), rmsg[0], rmsg[1], rmsg[2]
Call: MsgSend,MsgSendv,MsgSendvs
#Args: MSG_SENDV, fEstatus, frmsg, s, s

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_SENDVNC
Fast: coid, msg
Wide: coid, sparts, rparts, msg[0], msg[1], msg[2]
Call: MsgSendnc,MsgSendvnc,MsgSendvsnc
#Args: MSG_SENDVNC, fHcoid, Dsparts, Drparts, fSmsg, s, s

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_SENDVNC|_NTO_TRACE_KERCALL64
Fast: coid, msg
Wide: coid, sparts (64), rparts (64), msg[0], msg[1], msg[2]
Call: MsgSendnc,MsgSendvnc,MsgSendvsnc
#Args: MSG_SENDVNC, fHcoid, Esparts, Erparts, fSmsg, s, s

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_SENDVNC
Fast: ret_val, rmsg[0]
Wide: ret_val, rmsg[0], rmsg[1], rmsg[2]
Call: MsgSendnc,MsgSendvnc,MsgSendvsnc
#Args: MSG_SENDVNC, fHret_val, fSrmsg, s, s

Class: _NTO_TRACE_KERCALLEXIT
```

```
Event: __KER_MSG_SENDVNC|_NTO_TRACE_KERCALL64
Fast: ret_val (64)
Wide: ret_val (64), rmsg[0], rmsg[1], rmsg[2]
Call: MsgSendnc,MsgSendvnc,MsgSendvsnc
#Args: MSG_SENDVNC, fXret_val, fSrmsg, s, s

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_VERIFY_EVENT
Fast: rcvid, event->sigev_notify
Wide: rcvid, event->sigev_notify, event->sigev_notify_function_p (64), event->sigev_value (64),
event->sigev_notify_attribute_p (64)
Call: MsgVerifyEvent

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_VERIFY_EVENT
Fast: status, empty
Wide: status, empty
Call: MsgVerifyEvent
#Args: MSG_VERIFY_EVENT, fDstatus, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_WRITEV
Fast: rcvid, offset
Wide: rcvid, sparts, offset, msg[0], msg[1], msg[2]
Call: MsgWrite,MsgWritev
#Args: MSG_WRITEV, fHrcvid, Dsparts, fHoffset, Smsg, s, s

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_MSG_WRITEV|_NTO_TRACE_KERCALL64
Fast: rcvid, empty
Wide: rcvid, sparts (64), offset (64), msg[0], msg[1], msg[2]
Call: MsgWrite,MsgWritev
#Args: MSG_WRITEV, fHrcvid, Esparts, Xoffset, Smsg, s, s

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_WRITEV
Fast: wbytes, empty
Wide: wbytes, empty
Call: MsgWrite,MsgWritev
#Args: MSG_WRITEV, fDwbytes, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_MSG_WRITEV|_NTO_TRACE_KERCALL64
Fast: wbytes (64)
Wide: wbytes (64)
Call: MsgWrite,MsgWritev
#Args: MSG_WRITEV, fEwbytes

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_NET_CRED
Fast: coid, info_p
Wide: coid, info_p
Call: NetCred
#Args: NET_CRED, fHcoid, fPinfo_p
```

```
Class: _NTO_TRACE_KERCALLENTER
Event: __KER_NET_CRED|_NTO_TRACE_KERCALL64
Fast: coid
Wide: coid
Call: NetCred
#Args: NET_CRED, fHcoid, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_NET_CRED
Fast: ret_val, info->nd
Wide: ret_val, info->nd, info->pid, info->sid, info->flags, info->ruid, info->euid, info->suid, info->rgid,
info->egid, info->sgid, info->ngroups, info->grouplist[0], info->grouplist[1], info->grouplist[2],
info->grouplist[3], info->grouplist[4], info->grouplist[5], info->grouplist[6], info->grouplist[7]
Call: NetCred

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_NET_INFOSCOID
Fast: scoid, infoscoid
Wide: scoid, infoscoid
Call: NetInfoScoid
#Args: NET_INFOSCOID, fHscoid, fHinfoscoid

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_NET_INFOSCOID
Fast: ret_val, empty
Wide: ret_val, empty
Call: NetInfoScoid
#Args: NET_INFOSCOID, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_NET_SIGNAL_KILL
Fast: pid, signo
Wide: cred->ruid, cred->euid, nd, pid, tid, signo, code, value
Call: NetSignalKill
#Args: NET_SIGNAL_KILL, fDstatus, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_NET_SIGNAL_KILL
Fast: status, empty
Wide: status, empty
Call: NetSignalKill
#Args: NET_SIGNAL_KILL, fDstatus, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_NET_UNBLOCK
Fast: vtid, empty
Wide: vtid, empty
Call: NetUnblock
#Args: NET_UNBLOCK, fHvtid, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_NET_UNBLOCK
Fast: ret_val, empty
```

```
Wide: ret_val, empty
Call: NetUnblock
#Args: NET_UNBLOCK, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_NET_VTID
Fast: vtid, info_p
Wide: vtid, info_p, tid, coid, priority, srcmsglen, keydata, srcnd, dstmsglen
Call: NetVtid

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_NET_VTID|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: vtid, info_p (64), tid, coid, priority, srcmsglen, keydata, srcnd, dstmsglen
Call: NetVtid

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_NET_VTID
Fast: ret_val, empty
Wide: ret_val, empty
Call: NetVtid
#Args: NET_VTID, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_NOP
Fast: dummy, empty
Wide: dummy, empty
Call: N/A
#Args: NOP, fHdummy, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_NOP
Fast: empty, empty
Wide: empty, empty
Call: N/A
#Args: NOP, fHempty, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_RING0
Fast: func_p, arg_p
Wide: func_p, arg_p
Call: __Ring0
#Args: RING0, fPfunc_p, fParg_p

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_RING0|_NTO_TRACE_KERCALL64
Fast: func_p (64)
Wide: func_p (64)
Call: __Ring0
#Args: RING0, fQfunc_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_RING0
Fast: ret_val, empty
```

```
Wide: ret_val, empty
Call: __Ring0
#Args: RING0, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SCHED_GET
Fast: pid, tid
Wide: pid, tid
Call: SchedGet
#Args: SCHED_GET, fDpid, fDtid

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SCHED_GET
Fast: ret_val, sched_priority
Wide: ret_val, sched_priority, param.ss_low_priority, param.ss_max_repl, param.ss_repl_period.tv_sec,
param.ss_repl_period.tv_nsec, param.ss_init_budget.tv_sec, param.ss_init_budget.tv_nsec
Call: SchedGet
#Args: SCHED_GET, fDpid, fDtid

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SCHED_GET|_NTO_TRACE_KERCALL64
Fast: ret_val, sched_priority
Wide: ret_val, sched_priority, param.ss_low_priority, param.ss_max_repl, param.ss_repl_period.tv_sec (64),
param.ss_repl_period.tv_nsec (64), param.ss_init_budget.tv_sec (64), param.ss_init_budget.tv_nsec (64)
Call: SchedGet
#Args: SCHED_GET, fDpid, fDtid

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SCHED_INFO
Fast: pid, policy
Wide: pid, policy
Call: SchedInfo
#Args: SCHED_INFO, fDpid, fDpolicy

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SCHED_INFO
Fast: ret_val, priority_max
Wide: ret_val, priority_min, priority_max, interval_sec, interval_nsec, priority_priv
Call: SchedInfo
#Args: SCHED_INFO, fDpid, fDpolicy

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SCHED_SET
Fast: pid, sched_priority
Wide: pid, tid, policy, sched_priority, sched_curpriority, param.ss_low_priority, param.ss_max_repl,
param.ss_repl_period.tv_sec, param.ss_repl_period.tv_nsec, param.ss_init_budget.tv_sec,
param.ss_init_budget.tv_nsec
Call: SchedSet
#Args: SCHED_SET, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SCHED_SET|_NTO_TRACE_KERCALL64
Fast: pid, sched_priority
Wide: pid, tid, policy, sched_priority, sched_curpriority, param.ss_low_priority, param.ss_max_repl,
```

```
param.ss_repl_period.tv_sec (64), param.ss_repl_period.tv_nsec (64), param.ss_init_budget.tv_sec (64),
param.ss_init_budget.tv_nsec (64)
Call: SchedSet
#Args: SCHED_SET, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SCHED_SET
Fast: ret_val, empty
Wide: ret_val, empty
Call: SchedSet
#Args: SCHED_SET, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SCHED_YIELD
Fast: empty, empty
Wide: empty, empty
Call: SchedYield
#Args: SCHED_YIELD, fHempty, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SCHED_YIELD
Fast: ret_val, empty
Wide: ret_val, empty
Call: SchedYield
#Args: SCHED_YIELD, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SIGNAL_FAULT
Fast: sigcode, addr
Wide: sigcode, addr
Call: N/A
#Args: SIGNAL_FAULT, fDsigcode, fPaddr

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SIGNAL_FAULT|_NTO_TRACE_KERCALL64
Fast: sigcode
Wide: sigcode
Call: N/A
#Args: SIGNAL_FAULT, fDsigcode, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SIGNAL_FAULT
Fast: ret_val, reg_1
Wide: ret_val, reg_1, reg_2, reg_3, reg_4, reg_5
Call: N/A
#Args: SIGNAL_FAULT, fHret_val, fHreg_1, Hreg_2, Hreg_3, Hreg_4, Hreg_5

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SIGNAL_KILL
Fast: pid, signo
Wide: nd, pid, tid, signo, code, value
Call: SignalKill
#Args: SIGNAL_KILL, Hnd, fDpid, Dtid, fDsigno, Hcode, Hvalue
```

```
Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SIGNAL_KILL
Fast: ret_val, empty
Wide: ret_val, empty
Call: SignalKill
#Args: SIGNAL_KILL, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SIGNAL_KILL_SIGVAL
Fast: pid, signo
Wide: nd, pid, tid, signo, code, value
Call: SignalKill
#Args: SIGNAL_KILL_SIGVAL, Hnd, fDpid, Dtid, fDsigno, Hcode, Hvalue

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SIGNAL_KILL_SIGVAL|_NTO_TRACE_KERCALL64
Fast: pid, signo
Wide: nd, pid, tid, signo, code, value (64)
Call: SignalKillSigval
#Args: SIGNAL_KILL_SIGVAL, Hnd, fDpid, Dtid, fDsigno, Hcode, Xvalue

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SIGNAL_KILL_SIGVAL
Fast: ret_val, empty
Wide: ret_val, empty
Call: SignalKillSigval
#Args: SIGNAL_KILL_SIGVAL, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SIGNAL_RETURN
Fast: s_p, empty
Wide: s_p, empty
Call: SignalReturn
#Args: SIGNAL_RETURN, fPs_p, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SIGNAL_RETURN|_NTO_TRACE_KERCALL64
Fast: s_p (64)
Wide: s_p (64)
Call: SignalReturn
#Args: SIGNAL_RETURN, fQs_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SIGNAL_RETURN
Fast: ret_val, empty
Wide: ret_val, empty
Call: SignalReturn
#Args: SIGNAL_RETURN, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SIGNAL_SUSPEND
Fast: sig_blocked->bits[0], sig_blocked->bits[1]
Wide: sig_blocked->bits[0], sig_blocked->bits[1]
Call: SignalSuspend
```

```
#Args: SIGNAL_SUSPEND, fHsig_blocked->bits[0], fHsig_blocked->bits[1]

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SIGNAL_SUSPEND
Fast: ret_val, sig_blocked_p
Wide: ret_val, sig_blocked_p
Call: SignalSuspend
#Args: SIGNAL_SUSPEND, fHret_val, fPsig_blocked_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SIGNAL_SUSPEND
Fast: N/A
Wide: ret_val, sig_blocked_p (64)
Call: SignalSuspend
#Args: SIGNAL_SUSPEND, Hret_val, Qsig_blocked_p

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SIGNAL_WAITINFO
Fast: sig_wait->bits[0], sig_wait->bits[1]
Wide: sig_wait->bits[0], sig_wait->bits[1]
Call: SignalWaitinfo
#Args: SIGNAL_WAITINFO, fHsig_wait->bits[0], fHsig_wait->bits[1]

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SIGNAL_WAITINFO
Fast: sig_num, si_code
Wide: sig_num, si_signo, si_code, si_errno, p[0], p[1], p[2], p[3], p[4], p[5], p[6]
Call: SignalWaitinfo
#Args: SIGNAL_WAITINFO, fHsig_wait->bits[0], fHsig_wait->bits[1]

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_CONDVAR_SIGNAL
Fast: sync_p, all
Wide: sync_p, all, sync->count, sync->owner
Call: SyncCondvarSignal
#Args: SYNC_CONDVAR_SIGNAL, fPsync_p, fDall, Dsync->count, Dsync->owner

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_CONDVAR_SIGNAL|_NTO_TRACE_KERCALL64
Fast: sync_p (64)
Wide: sync_p (64), all, sync->count, sync->owner
Call: SyncCondvarSignal
#Args: SYNC_CONDVAR_SIGNAL, fQsync_p, Dall, Dsync->count, Dsync->owner

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SYNC_CONDVAR_SIGNAL
Fast: ret_val, empty
Wide: ret_val, empty
Call: SyncCondvarSignal
#Args: SYNC_CONDVAR_SIG, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_CONDVAR_WAIT
Fast: sync_p, mutex_p
```

```
Wide: sync_p, mutex_p, sync->count, sync->owner, mutex->count, mutex->owner
Call: SyncCondvarWait
#Args: SYNC_CONDVAR_WAIT, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_CONDVAR_WAIT|_NTO_TRACE_KERCALL64
Fast: sync_p (64)
Wide: sync_p (64), mutex_p (64), sync->count, sync->owner, mutex->count, mutex->owner
Call: SyncCondvarWait
#Args: SYNC_CONDVAR_WAIT, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SYNC_CONDVAR_WAIT
Fast: ret_val, empty
Wide: ret_val, empty
Call: SyncCondvarWait
#Args: SYNC_CONDVAR_WAIT, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_CREATE
Fast: type, sync_p
Wide: type, sync_p, count, owner, protocol, flags, prioceiling, clockid
Call: SyncCreate
#Args: SYNC_CREATE, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_CREATE|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: type, sync_p (64), count, owner, protocol, flags, prioceiling, clockid
Call: SyncCreate
#Args: SYNC_CREATE, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SYNC_CREATE
Fast: ret_val, empty
Wide: ret_val, empty
Call: SyncCreate
#Args: SYNC_CREATE, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_CTL
Fast: cmd, sync_p
Wide: cmd, sync_p, data_p, count, owner
Call: SyncCtl
#Args: SYNC_CTL, fDcmd, fPsync_p, Pdata_p, Dcount, Downer

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_CTL|_NTO_TRACE_KERCALL64
Fast: cmd
Wide: cmd, sync_p (64), data_p (64), count, owner
Call: SyncCtl
#Args: SYNC_CTL, fDcmd, Qsync_p, Qdata_p, Dcount, Downer

Class: _NTO_TRACE_KERCALLEXIT
```

```
Event: __KER_SYNC_CTL
Fast: ret_val, empty
Wide: ret_val, empty
Call: SyncCtl
#Args: SYNC_CTL, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_DESTROY
Fast: sync_p, owner
Wide: sync_p, count, owner
Call: SyncDestroy
#Args: SYNC_DESTROY, fPsync_p, Dcount, fDowner

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_DESTROY|_NTO_TRACE_KERCALL64
Fast: sync_p (64)
Wide: sync_p (64), count, owner
Call: SyncDestroy
#Args: SYNC_DESTROY, fQsync_p, Dcount, Downer

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SYNC_DESTROY
Fast: ret_val, empty
Wide: ret_val, empty
Call: SyncDestroy
#Args: SYNC_DESTROY, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_MUTEX_LOCK
Fast: sync_p, owner
Wide: sync_p, count, owner
Call: SyncMutexLock
#Args: SYNC_MUTEX_LOCK, fPsync_p, Dcount, fDowner

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_MUTEX_LOCK|_NTO_TRACE_KERCALL64
Fast: sync_p (64)
Wide: sync_p (64), count, owner
Call: SyncMutexLock
#Args: SYNC_MUTEX_LOCK, fQsync_p, Dcount, Downer

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SYNC_MUTEX_LOCK
Fast: ret_val, empty
Wide: ret_val, empty
Call: SyncMutexLock
#Args: SYNC_MUTEX_LOCK, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_MUTEX_REVIVE
Fast: sync_p, owner
Wide: sync_p, count, owner
Call: SyncMutexRevive
#Args: SYNC_MUTEX_REVIVE, fPsync_p, Dcount, fDowner
```

```
Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_MUTEX_REVIVE|_NTO_TRACE_KERCALL64
Fast: sync_p (64)
Wide: sync_p (64), count, owner
Call: SyncMutexRevive
#Args: SYNC_MUTEX_REVIVE, fQsync_p, Dcount, Downer

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SYNC_MUTEX_REVIVE
Fast: ret_val, empty
Wide: ret_val, empty
Call: SyncMutexRevive
#Args: SYNC_MUTEX_REVIVE, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_MUTEX_UNLOCK
Fast: sync_p, owner
Wide: sync_p, count, owner
Call: SyncMutexUnlock
#Args: SYNC_MUTEX_UNLOCK, fPsync_p, Dcount, fDowner

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_MUTEX_UNLOCK|_NTO_TRACE_KERCALL64
Fast: sync_p (64), owner
Wide: sync_p (64), count, owner
Call: SyncMutexUnlock
#Args: SYNC_MUTEX_UNLOCK, fQsync_p, Dcount, Downer

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SYNC_MUTEX_UNLOCK
Fast: ret_val, empty
Wide: ret_val, empty
Call: SyncMutexUnlock
#Args: SYNC_MUTEX_UNLOCK, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_SEM_POST
Fast: sync_p, count
Wide: sync_p, count, owner
Call: SyncSemPost
#Args: SYNC_SEM_POST, fPsync_p, fDcount, Downer

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_SEM_POST|_NTO_TRACE_KERCALL64
Fast: sync_p (64)
Wide: sync_p (64), count, owner
Call: SyncSemPost
#Args: SYNC_SEM_POST, fQsync_p, Dcount, Downer

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SYNC_SEM_POST
Fast: ret_val, empty
Wide: ret_val, empty
```

```
Call: SyncSemPost
#Args: SYNC_SEM_POST, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_SEM_WAIT
Fast: sync_p, count
Wide: sync_p, try, count, owner
Call: SyncSemWait
#Args: SYNC_SEM_WAIT, fPsync_p, Dtry, fDcount, Downer

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYNC_SEM_WAIT|_NTO_TRACE_KERCALL64
Fast: sync_p (64)
Wide: sync_p (64), try, count, owner
Call: SyncSemWait
#Args: SYNC_SEM_WAIT, fQsync_p, Dtry, Dcount, Downer

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SYNC_SEM_WAIT
Fast: ret_val, empty
Wide: ret_val, empty
Call: SyncSemWait
#Args: SYNC_SEM_WAIT, fDret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYS_CPUPAGE_GET
Fast: index, empty
Wide: index, empty
Call: N/A
#Args: SYS_CPUPAGE_GET, fDindex, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SYS_CPUPAGE_GET
Fast: ret_val, empty
Wide: ret_val, empty
Call: N/A
#Args: SYS_CPUPAGE_GET, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_THREAD_CANCEL
Fast: tid, canstub_p
Wide: tid, canstub_p
Call: ThreadCancel
#Args: THREAD_CANCEL, fDtid, fPcanstub_p

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_THREAD_CANCEL|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: tid, canstub_p (64)
Call: ThreadCancel
#Args: THREAD_CANCEL, Dtid, Qcanstub_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_THREAD_CANCEL
```

```
Fast: ret_val, empty
Wide: ret_val, empty
Call: ThreadCancel
#Args: THREAD_CANCEL, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_THREAD_CREATE
Fast: func_p, arg_p
Wide: pid, func_p, arg_p, flags, stacksize, stackaddr_p, exitfunc_p, policy, sched_priority,
sched_curpriority, param.ss_low_priority, param.ss_max_repl, param.ss_repl_period.tv_sec,
param.ss_repl_period.tv_nsec, param.ss_init_budget.tv_sec, param.ss_init_budget.tv_nsec
Call: ThreadCreate
#Args: THREAD_CREATE, fHthread_id, fHowner

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_THREAD_CREATE|_NTO_TRACE_KERCALL64
Fast: func_p (64)
Wide: pid, func_p (64), arg_p (64), flags, pad, stacksize (64), stackaddr_p (64), exitfunc_p (64), policy,
sched_priority, sched_curpriority, param.ss_low_priority, param.ss_max_repl, param.ss_repl_period.tv_sec (64),
param.ss_repl_period.tv_nsec (64), param.ss_init_budget.tv_sec (64), param.ss_init_budget.tv_nsec (64)
Call: ThreadCreate
#Args: THREAD_CREATE, fHthread_id, fHowner

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_THREAD_CREATE
Fast: thread_id, owner
Wide: thread_id, owner
Call: ThreadCreate
#Args: THREAD_CREATE, fHthread_id, fHowner

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_THREAD_CTL
Fast: cmd, data_p
Wide: cmd, data_p
Call: ThreadCtl
#Args: THREAD_CTL, fHcmd, fPdata_p

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_THREAD_CTL|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: cmd, data_p
Call: ThreadCtl
#Args: THREAD_CTL, Hcmd, Qdata_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_THREAD_CTL
Fast: ret_val, empty
Wide: ret_val, empty
Call: ThreadCtl
#Args: THREAD_CTL, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_THREAD_DESTROY
Fast: tid, status_p
```



```
Wide: tid, priority, status_p
Call: ThreadDestroy
#Args: THREAD_DESTROY, fDtid, Dpriority, fPstatus_p

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_THREAD_DESTROY|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: tid, priority, status_p (64)
Call: ThreadDestroy
#Args: THREAD_DESTROY, Dtid, Dpriority, Qstatus_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_THREAD_DESTROY
Fast: ret_val, empty
Wide: ret_val, empty
Call: ThreadDestroy
#Args: THREAD_DESTROY, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_THREAD_DESTROYALL
Fast: empty, empty
Wide: empty, empty
Call: N/A
#Args: THREAD_DESTROYALL, fHempty, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_THREAD_DESTROYALL
Fast: ret_val, empty
Wide: ret_val, empty
Call: N/A
#Args: THREAD_DESTROYALL, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_THREAD_DETACH
Fast: tid, empty
Wide: tid, empty
Call: ThreadDetach
#Args: THREAD_DETACH, fDtid, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_THREAD_DETACH
Fast: ret_val, empty
Wide: ret_val, empty
Call: ThreadDetach
#Args: THREAD_DETACH, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_THREAD_JOIN
Fast: tid, status_p
Wide: tid, status_p
Call: ThreadJoin
#Args: THREAD_JOIN, fDtid, fPstatus_p

Class: _NTO_TRACE_KERCALLENTER
```

```
Event: __KER_THREAD_JOIN|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: tid, status_p (64)
Call: ThreadJoin
#Args: THREAD_JOIN, Dtid, Qstatus_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_THREAD_JOIN
Fast: ret_val, status_p
Wide: ret_val, status_p
Call: ThreadJoin
#Args: THREAD_JOIN, fHret_val, fPstatus_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_THREAD_JOIN|_NTO_TRACE_KERCALL64
Fast: N/A
Wide: ret_val, status_p (64)
Call: ThreadJoin
#Args: THREAD_JOIN, Hret_val, Qstatus_p

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_TIMER_CREATE
Fast: timer_id, event->sigev_notify
Wide: timer_id, event->sigev_notify, event->sigev_notify_function_p, event->sigev_value,
event->sigev_notify_attributes_p
Call: TimerCreate
#Args: TIMER_CREATE, fHtimer_id, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_TIMER_CREATE|_NTO_TRACE_KERCALL64
Fast: timer_id, event->sigev_notify
Wide: timer_id, event->sigev_notify, pad, event->sigev_notify_function_p (64), event->sigev_value (64),
event->sigev_notify_attributes_p (64)
Call: TimerCreate
#Args: TIMER_CREATE, fHtimer_id, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_TIMER_CREATE
Fast: timer_id, empty
Wide: timer_id, empty
Call: TimerCreate
#Args: TIMER_CREATE, fHtimer_id, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_TIMER_DESTROY
Fast: id, empty
Wide: id, empty
Call: TimerDestroy
#Args: TIMER_DESTROY, fHid, fHempty

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_TIMER_DESTROY
Fast: ret_val, empty
Wide: ret_val, empty
```

```
Call: TimerDestroy
#Args: TIMER_DESTROY, fHret_val, fHempty

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_TIMER_INFO
Fast: pid, id
Wide: pid, id, flags, info_p
Call: TimerInfo
#Args: TIMER_INFO, fDpid, fHid, Hflags, Pinfo_p

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_TIMER_INFO|_NTO_TRACE_KERCALL64
Fast: pid, id
Wide: pid, id, flags, info_p (64)
Call: TimerInfo
#Args: TIMER_INFO, fDpid, fHid, Hflags, Qinfo_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_TIMER_INFO
Fast: prev_id, info->itime.nsec
Wide: prev_id, info->itime.nsec, info->itime.interval_nsec, info->otime.nsec, info->otime.interval_nsec,
info->flags, info->tid, info->notify, info->clockid, info->overruns, info->event.sigev_notify,
info->event.sigev_notify_function_p, info->event.sigev_value, info->event.sigev_notify_attributes_p
Call: TimerInfo
#Args: TIMER_INFO, fDpid, fHid, Hflags, Pinfo_p

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_TIMER_INFO|_NTO_TRACE_KERCALL64
Fast: prev_id, info->itime.nsec
Wide: prev_id, info->itime.nsec, info->itime.interval_nsec, info->otime.nsec, info->otime.interval_nsec,
info->flags, info->tid, info->notify, info->clockid, info->overruns, info->event.sigev_notify, pad,
info->event.sigev_notify_function_p (64), info->event.sigev_value (64),
info->event.sigev_notify_attributes_p (64)
Call: TimerInfo
#Args: TIMER_INFO, fDpid, fHid, Hflags, Pinfo_p

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_TRACE_EVENT
Fast: mode, class
Wide: mode, class, event, data_1, data_2
Call: TraceEvent
#Args: TRACE_EVENT, fHmode, fHclass[header], Hevent[time_off], Hdata_1, Hdata_2

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_TRACE_EVENT|_NTO_TRACE_KERCALL64
Fast: mode, class
Wide: mode (64), class (64), event (64), data_1 (64), data_2 (64)
Call: TraceEvent
#Args: TRACE_EVENT, fXmode, fXclass[header], Xevent[time_off], Xdata_1, Xdata_2

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_TRACE_EVENT
Fast: ret_val, empty
Wide: ret_val, empty
```

```
Call: TraceEvent
#Args: TRACE_EVENT, fHret_val, fHempty

Class: _NTO_TRACE_INTENTER
Event: _NTO_TRACE_INTFIRST - _NTO_TRACE_INTLAST
Fast: IP, kernel_flag
Wide: interrupt_number, kernel_flag
Call: N/A

Class: _NTO_TRACE_INTEXIT
Event: _NTO_TRACE_INTFIRST - _NTO_TRACE_INTLAST
Fast: interrupt_number, kernel_flag
Wide: interrupt_number, kernel_flag
Call: N/A

Class: _NTO_TRACE_INT_HANDLER_ENTER
Event: _NTO_TRACE_INTFIRST - _NTO_TRACE_INTLAST
Fast: pid, interrupt_number, ip, area
Wide: pid, interrupt_number, ip, area
Call: N/A

Class: _NTO_TRACE_INT_HANDLER_EXIT
Event: _NTO_TRACE_INTFIRST - _NTO_TRACE_INTLAST
Fast: interrupt_number, sigevent
Wide: interrupt_number, sigevent
Call: N/A

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SIGNAL_ACTION
Fast: signo, act->sa_handler_p
Wide: pid, sigstub_p, signo, act->sa_handler_p, act->sa_flags, act->sa_mask.bits[0], act->sa_mask.bits[1]
Call: SignalAction

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SIGNAL_ACTION|_NTO_TRACE_KERCALL64
Fast: signo
Wide: pid, sigstub_p (64), signo, act->sa_handler_p (64), act->sa_flags, act->sa_mask.bits[0],
act->sa_mask.bits[1]
Call: SignalAction

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SIGNAL_ACTION
Fast: ret_val, act->sa_handler_p
Wide: ret_val, act->sa_handler_p, act->sa_flags, act->sa_mask.bits[0], act->sa_mask.bits[1]
Call: SignalAction

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SIGNAL_ACTION|_NTO_TRACE_KERCALL64
Fast: ret_val
Wide: ret_val, act->sa_handler_p (64), act->sa_flags, act->sa_mask.bits[0],
act->sa_mask.bits[1]
Call: SignalAction

Class: _NTO_TRACE_KERCALLENTER
```

```
Event: __KER_SIGNAL_PROCMASK
Fast: pid, tid
Wide: pid, tid, how, sig_blocked->bits[0], sig_blocked->bits[1]
Call: SignalProcmask

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SIGNAL_PROCMASK
Fast: ret_val, sig_blocked->bits[0]
Wide: ret_val, sig_blocked->bits[0], sig_blocked->bits[1]
Call: SignalProcmask

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_TIMER_SETTIME
Fast: clock_id, itime->nsec
Wide: clock_id, flags, itime->nsec, itime->interval_nsec
Call: TimerSettime

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_TIMER_SETTIME
Fast: ret_val, itime->nsec
Wide: ret_val, itime->nsec, itime->interval_nsec
Call: TimerSettime

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_TIMER_ALARM
Fast: clock_id, itime->nsec
Wide: clock_id, itime->nsec, itime->interval_nsec
Call: TimerAlarm

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_TIMER_ALARM
Fast: ret_val, itime->nsec
Wide: ret_val, itime->nsec, itime->interval_nsec
Call: TimerAlarm

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_TIMER_TIMEOUT
Fast: clock_id, timeout_flags, ntime
Wide: clock_id, timeout_flags, ntime, event->sigev_notify, event->sigev_notify_function_p,
event->sigev_value, event->sigev_notify_attributes_p
Call: TimerTimeout

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_TIMER_TIMEOUT|_NTO_TRACE_KERCALL64
Fast: clock_id, timeout_flags, ntime
Wide: clock_id, timeout_flags, ntime, event->sigev_notify, pad, event->sigev_notify_function_p (64),
event->sigev_value (64), event->sigev_notify_attributes_p (64)
Call: TimerTimeout

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_TIMER_TIMEOUT
Fast: prev_timeout_flags, otime
Wide: prev_timeout_flags, otime
Call: TimerTimeout
```



```
# Control Events

Class: _NTO_TRACE_CONTROL
Event: _NTO_TRACE_CONTROLTIME
Fast: msbtime, lsbtime
Wide: msbtime, lsbtime
Call: N/A

Class: _NTO_TRACE_CONTROL
Event: _NTO_TRACE_CONTROLBUFFER
Fast: buffer sequence number, num events
Wide: buffer sequence number, num events
Call: N/A

# Process Events

Class: _NTO_TRACE_PROCESS
Event: _NTO_TRACE_PROCCREATE
Fast: ppid, pid
Wide: ppid, pid
Call: N/A

Class: _NTO_TRACE_PROCESS
Event: _NTO_TRACE_PROCCREATE_NAME
Fast: ppid, pid, name
Wide: ppid, pid, name
Call: N/A

Class: _NTO_TRACE_PROCESS
Event: _NTO_TRACE_PROCDESTROY
Fast: ppid, pid
Wide: ppid, pid
Call: N/A

Class: _NTO_TRACE_PROCESS
Event: _NTO_TRACE_PROCDESTROY_NAME
Fast: ppid, pid, name
Wide: ppid, pid, name
Call: N/A

Class: _NTO_TRACE_PROCESS
Event: _NTO_TRACE_PROCTHREAD_NAME
Fast: pid, tid, name
Wide: pid, tid, name
Call: N/A

#Thread state changes

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THDEAD
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
```

```
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THRUNNING
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THREADY
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THSTOPPED
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THSEND
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THRECEIVE
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THREPLY
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THSTACK
Fast: pid, tid
```

```
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THWAITTHREAD
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THWAITPAGE
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THSIGSUSPEND
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THSIGWAITINFO
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THNANOSLEEP
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THMUTEX
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
```

```
Event: _NTO_TRACE_THCONDVAR
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THJOIN
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THINTR
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THSEM
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THWAITCTX
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THNET_SEND
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THNET_REPLY
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A
```

```
Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THCREATE
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THDESTROY
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

Class: _NTO_TRACE_THREAD
Event: _NTO_TRACE_THNET_REPLY
Fast: pid, tid
Wide: pid, tid, priority, policy, partition id, sched_flags (incl APS_SCHED_* critical bit def'd
in kermacros.h)
 ** note: partition id and sched_flags only present if APS scheduler module is loaded.
Call: N/A

#VThread state changes

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHDEAD
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHRUNNING
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHREADY
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHSTOPPED
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHSEND
Fast: pid, tid
```



```
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHRECEIVE
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHREPLY
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHSTACK
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHWAITTHREAD
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHWAITPAGE
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHSIGSUSPEND
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHSIGWAITINFO
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHNANOSLEEP
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHMUTEX
Fast: pid, tid
```

```
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHCONDVAR
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHJOIN
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHINTR
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHSEM
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHWAITCTX
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHNET_SEND
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHNET_REPLY
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHCREATE
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHDESTROY
Fast: pid, tid
```

```
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_VTHREAD
Event: _NTO_TRACE_VTHNET_REPLY
Fast: pid, tid
Wide: pid, tid
Call: N/A

Class: _NTO_TRACE_COMM
Event: _NTO_TRACE_COMM_SMSG
Fast: rcvid, pid
Wide: rcvid, pid
Call: N/A

Class: _NTO_TRACE_COMM
Event: _NTO_TRACE_COMM_RMSG
Fast: rcvid, pid
Wide: rcvid, pid
Call: N/A

Class: _NTO_TRACE_COMM
Event: _NTO_TRACE_COMM_REPLY
Fast: tid, pid
Wide: tid, pid
Call: N/A

Class: _NTO_TRACE_COMM
Event: _NTO_TRACE_COMM_ERROR
Fast: tid, pid
Wide: tid, pid
Call: N/A

Class: _NTO_TRACE_COMM
Event: _NTO_TRACE_COMM_SPULSE
Fast: scoid, pid
Wide: scoid, pid
Call: N/A

Class: _NTO_TRACE_COMM
Event: _NTO_TRACE_COMM_RPULSE
Fast: scoid, pid
Wide: scoid, pid
Call: N/A

# SIGEV_PULSE delivered
Class: _NTO_TRACE_COMM
Event: _NTO_TRACE_COMM_SPULSE_EXE
Fast: scoid, pid
Wide: scoid, pid
Call: N/A

# _PULSE_CODE_DISCONNECT pulse delivered
Class: _NTO_TRACE_COMM
```

```
Event: _NTO_TRACE_COMM_SPULSE_DIS
Fast: scoid, pid
Wide: scoid, pid
Call: N/A

# _PULSE_CODE_COIDDEATH pulse delivered
Class: _NTO_TRACE_COMM
Event: _NTO_TRACE_COMM_SPULSE_DEA
Fast: scoid, pid
Wide: scoid, pid
Call: N/A

# _PULSE_CODE_UNBLOCK delivered
Class: _NTO_TRACE_COMM
Event: _NTO_TRACE_COMM_SPULSE_UN
Fast: scoid, pid
Wide: scoid, pid
Call: N/A

# _PULSE_CODE_NET_UNBLOCK delivered
Class: _NTO_TRACE_COMM
Event: _NTO_TRACE_COMM_SPULSE_QUN
Fast: scoid, pid
Wide: scoid, pid
Call: N/A

Class: _NTO_TRACE_COMM
Event: _NTO_TRACE_COMM_SIGNAL
Fast: si_signo, si_code
Wide: si_signo, si_code, si_errno, __data.__pad[0-6]
Call: N/A

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_PATHMGR
Fast: pid, tid, pathname
Wide: pid, tid, pathname
Call: Any pathname operation (routed via libc connect)

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_APS_NAME
Fast: partition id, partition name
Wide: partition id, partition name
Call: SchedCtl with sched_aps.h:: SCHED_APS_CREATE_PARTITION

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_APS_BUDGETS
Fast: partition id, new percentage cpu budget, new critical budget ms
Wide: partition id, new percentage cpu budget, new critical budget ms
Call: SchedCtl with sched_aps.h SCHED_APS_CREATE_PARTITION or SCHED_APS_MODIFY_PARTITION. Also emitted
automatically when APS scheduler clears a critical budget as part of handling a bankruptcy.

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_APS_BNKR
Fast: suspect pid, suspect tid, partition id
```

```
Wide: suspect pid, suspect tid, partition id
Call: automatically when a partition exceeds its critical budget.

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_MMAP
Fast: pid, addr (64), len (64), flags
Wide: pid, addr (64), len (64), flags, prot, fd, align (64), offset (64), name
Call: mmap/mmap64

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_MUNMAP
Fast: pid, addr (64), len (64)
Wide: pid, addr (64), len (64)
Call: munmap

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_MAPNAME
Fast: pid, addr (32), len (32), name
Wide: pid, addr (32), len (32), name
Call: dlopen

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_MAPNAME_64
Fast: pid, addr (64), len (64), name
Wide: pid, addr (64), len (64), name
Call: dlopen

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_ADDRESS
Fast: addr(32), <null>
Wide: addr(32), <null>
Call: whenever a breakpoint is hit

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_FUNC_ENTER
Fast: thisfn(32), call_site(32)
Wide: thisfn(32), call_site(32)
Call: whenever a function is entered (and it is instrumented)

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_FUNC_ENTER|_NTO_TRACE_KERCALL64
Fast: thisfn(64), call_site(64)
Wide: thisfn(64), call_site(64)
Call: whenever a function is entered (and it is instrumented)

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_FUNC_EXIT
Fast: thisfn(32), call_site(32)
Wide: thisfn(32), call_site(32)
Call: whenever a function is exited (and it is instrumented)

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_FUNC_EXIT|_NTO_TRACE_KERCALL64
Fast: thisfn(64), call_site(64)
```

```
Wide: thisfn(64), call_site(64)
Call: whenever a function is exited (and it is instrumented)

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_SLOG
Fast: opcode(32), severity(32), message
Wide: opcode(32), severity(32), message
Call: when the kernel wants to note an unusual occurrence



Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_RUNSTATE
Fast: bitset(32) 0x1 - CPU is on/offline, 0x2 - CPU manually requested on/offline, 0x4 CPU is dynamically
offline-able or not, 0x8 system is in runstate burst mode
Wide: same as above
Call: when the runstate for a CPU changes

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_POWER
Fast: bitset(32), mode(32)
Wide: same as above
Call: idle mode entry/exit, CPU frequency change

    The bitset field holds the following:

 /* Power event flags */
 #define _NTO_TRACE_POWER_CPUMASK  0x0000ffffu
 #define _NTO_TRACE_POWER_IDLE   0x00010000u
 #define _NTO_TRACE_POWER_START   0x00020000u
 #define _NTO_TRACE_POWER_IDLE_REACHED 0x00040000u /* for _NTO_TRACE_POWER_IDLE */
 #define _NTO_TRACE_POWER_VFS_OVERDRIVE 0x00040000u /* for !_NTO_TRACE_POWER_IDLE */
 #define _NTO_TRACE_POWER_VFS_DYNAMIC 0x00080000u /* for !_NTO_TRACE_POWER_IDLE */
 #define _NTO_TRACE_POWER_VFS_STEP_UP 0x00100000u /* for !_NTO_TRACE_POWER_IDLE */

 The bottom 16 bits is the CPU that the mode change applies to. For
 idle events, this will always be the same as the CPU in the event header.
 For frequency changes, they may be different (e.g. CPU 0 changes CPU 1's
 frequency).

 If the POWER_IDLE bit is on, this an idle event, if off the event is a
 frequency change.

 If the POWER_START bit is on, it means that we're
 starting a power event: idle is being entered, we're kicking off a
 frequency change request. If the bit is off: we're coming out of
 idle, the frequency change has been completed.

 On the idle exit event, the IDLE_REACHED bit indicates that the CPU
 achieved the requested sleep mode.

 For frequency entry events, the VFS_OVERDRIVE bit indicates that the
 change was being requested by the reception of an overdrive
 sigevent. The VFS_DYNAMIC bit indicates that the DVFS algorithm is
```



```
 requesting a change due to CPU loading. If neither is on, it's a change
 due to powerman's list of allowed modes no longer including the
 frequency that we were previously running at.

 The second word of the event is the mode of the power event. For idle
 events, this is the number given by the "sleep=?" characteristic in the
 powerman configuration file. For frequency events, this is the
 value given by the "throughput=?" characteristic (usually the CPU
 frequency).

 Note that for frequency events, the second word for the entry
 event and exit event may be different. E.g. powerman might request
 CPU 0 to be run at 300MHz, but CPU 0 & CPU 1 frequencies are tied
 together and CPU 1 wants to run at 800MHz. In that case the CPU
 specific code may decide to run CPU 0 at 800MHz instead of the
 requested 300 and will report the fact in the exit event. Treat the
 frequency entry as the requested mode and the exit as the actual
 mode.

 Due to interrupt preemptions, you can not be guaranteed that for each
 entry event there will be a matching exit event and vice versa. E.g.
 there might be multiple idle entries before an idle exit or vice versa.

 Relatively shortly after the start of tracing, powerman will dump a
 series of frequency exit events giving the current frequencies of
 each of the CPU's. You should make the assumption that CPU was
 running in that mode at the start of the trace.

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_IPI
Fast: ipicmd(32), interrupted ip(32), tid(32), pid(32)
Wide: same as above
Call: when an inter-processor-interrupt is received

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_IPI_64
Fast: N/A
Wide: ipicmd(32), pad(32), interrupted ip(64), tid(32), pid(32)
Call: when an inter-processor-interrupt is received

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_PROFILE
Fast: ip(32), tid(32), pid(32)
Wide: same as above
Call: every clock tick, if statistical profiling is enabled

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_PROFILE_64
Fast: ip(64), tid(32), pid(32)
Wide: same as above
Call: every clock tick, if statistical profiling is enabled

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_PAGEWAIT
```

```
Fast: pid(32), tid(32), ip(32), vaddr(32)
Wide: pid(32), tid(32), ip(32), vaddr(32), fault type(32), mmap_flags(32), object_offset(64),
object_name(string)
Call: during page fault handling

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_SYS_TIMER
Fast: pid(32), tid(32), timer_id(32), flags(32)
Wide: same as above
Call: on timer expiry
Note - the timer_id will be -1 for timer_timeout expiry

Class: _NTO_TRACE_SYSTEM
Event: _NTO_TRACE_DEFRAG_END
Fast: rc(32), freemem(32), maxblock(32)
Wide: rc(32), freemem(32), maxblock(32)
Call: on completion of the defragmentation process (whether successful or not)

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SCHED_WAYPOINT
Fast: job, new, max, old, *new (64), *max (64)
Wide: same as above
Call: SchedWaypoint

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SCHED_WAYPOINT|_NTO_TRACE_KERCALL64
Fast: job (64), new (64), max (64), old (64), *new (64), *max (64)
Wide: same as above
Call: SchedWaypoint

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SCHED_WAYPOINT
Fast: ret_val, job, *old (64)
Wide: same as above
Call: SchedWaypoint

Class: _NTO_TRACE_KERCALLEXIT
Event: __KER_SCHED_WAYPOINT|_NTO_TRACE_KERCALL64
Fast: ret_val, job (64), *old (64)
Wide: same as above
Call: SchedWaypoint

Class: _NTO_TRACE_KERCALLENTER
Event: __KER_SYS_SRANDOM
Fast: N/A
Wide: N/A
Call: SysSrandom

# Security Events

Class: _NTO_TRACE_SEC
Event: _NTO_TRACE_SEC_ABLE
Fast: pid(32), able(32), flags(32), type_id(32), start(64), end(64)
Wide: same as above
```

```
Call: When a process is tested for having a procmgr ability

Class: _NTO_TRACE_SEC
Event: _NTO_TRACE_SEC_ABLE_LOOKUP
Fast: able(32), name
Wide: same as above
Call: When a non-predefined procmgr ability is looked up or created

Class: _NTO_TRACE_SEC
Event: _NTO_TRACE_SEC_PATH_ATTACH
Fast: pid(32), ptype(32), ctype(32), status(32), path
Wide: same as above
Call: When a link is made or failed to be made to the path space (resmgr_attach/pathmgr_link).

Class: _NTO_TRACE_SEC
Event: _NTO_TRACE_SEC_QNET_CONNECT
Fast: nd(32), pid(32), ptype(32), ctype(32), status(32)
Wide: same as above
Call: When a remote process attempts to connect to a channel over QNET and a security policy is in effect.
# Note this event is not emitted by procnto.
```

Index

D

E

F

H

I

P

Q

R

S

W