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
- 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.)

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 via messages and pulses:

This event:	Is emitted when:
_NTO_TRACE_COMM_ERROR	A client is unblocked because of a call to *MsgError()*
_NTO_TRACE_COMM_REPLY	A reply is sent
_NTO_TRACE_COMM_RMSG	A message is received
_NTO_TRACE_COMM_RPULSE	A pulse is received
_NTO_TRACE_COMM_SIGNAL	A signal is received
_NTO_TRACE_COMM_SMSG	A message is sent
_NTO_TRACE_COMM_SPULSE	A pulse is sent

continued...

This event:	Is emitted when:
_NTO_TRACE_COMM_SPULSE_DEA	A _PULSE_CODE_COIDDEATH pulse is sent
_NTO_TRACE_COMM_SPULSE_DIS	A _PULSE_CODE_DISCONNECT pulse is sent
_NTO_TRACE_COMM_SPULSE_EXE	A SIGEV_PULSE is sent
_NTO_TRACE_COMM_SPULSE_QUN	A _PULSE_CODE_NET_UNBLOCK pulse is sent
_NTO_TRACE_COMM_SPULSE_UN	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:

This event:	Is emitted when:
_NTO_TRACE_CONTROLBUFFER	The instrumented kernel starts filling a new buffer
_NTO_TRACE_CONTROLTIME	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:

- _NTO_TRACE_INTENTER and _NTO_TRACE_INTEXIT track the beginning and end of the overall processing of an interrupt.
- _NTO_TRACE_INT_HANDLER_ENTER and _NTO_TRACE_INT_HANDLER_EXIT track the entrances to and exits from interrupt handlers.

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.

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	N/A
__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()*

continued...

Event	Kernel call
__KER_INTERRUPT_ATTACH	*InterruptAttach()*
__KER_INTERRUPT_DETACH	*InterruptDetach()*
__KER_INTERRUPT_DETACH_FUNC	N/A
__KER_INTERRUPT_MASK	*InterruptMask()*
__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_READIOV	*MsgReadIov()*
__KER_MSG_READV	*MsgRead()*, *MsgReadv()*
__KER_MSG_READWRITEV	N/A
__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_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	N/A
__KER_RING0 (not generated in QNX Neutrino 6.3.0 or later)	*__Ring0()*

continued...

Event	Kernel call
__KER_SCHED_GET	*SchedGet()*
__KER_SCHED_INFO	*SchedInfo()*
__KER_SCHED_SET	*SchedSet()*
__KER_SCHED_YIELD	*SchedYield()*
__KER_SIGNAL_ACTION	*SignalAction()*
__KER_SIGNAL_FAULT	N/A
__KER_SIGNAL_KILL	*SignalKill()*
__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	N/A
__KER_THREAD_CANCEL	*ThreadCancel()*
__KER_THREAD_CREATE	*ThreadCreate()*
__KER_THREAD_CTL	*ThreadCtl()*
__KER_THREAD_DESTROY	*ThreadDestroy()*
__KER_THREAD_DESTROYALL	N/A
__KER_THREAD_DETACH	*ThreadDetach()*
__KER_THREAD_JOIN	*ThreadJoin()*

continued...

Event	Kernel call
__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:

This event:	Is emitted when:
_NTO_TRACE_PROCCREATE	A process is created
_NTO_TRACE_PROCCREATE_NAME	A newly created process is given a name.
_NTO_TRACE_PROCDESTROY	A process is destroyed
_NTO_TRACE_PROCDESTROY_NAME (not currently used)	N/A
_NTO_TRACE_PROCTHREAD_NAME	A name is assigned to a thread

System class: _NTO_TRACE_SYSTEM

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

This event:	Is emitted when:
_NTO_TRACE_SYS_ADDRESS	A breakpoint is hit
_NTO_TRACE_SYS_APS_BNKR	An adaptive partition exceeded its critical budget
_NTO_TRACE_SYS_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.

continued...

This event:	Is emitted when:
_NTO_TRACE_SYS_APS_NAME	*SchedCtl()* is called with a command of SCHED_APS_CREATE_PARTITION
_NTO_TRACE_SYS_COMPACTION	The memory defragmentation **`compaction_minimal`** algorithm is triggered (see "Defragmenting physical memory" in the Process Manager chapter of the *System Architecture* guide)
_NTO_TRACE_SYS_FUNC_ENTER	A function that's instrumented for profiling is entered
_NTO_TRACE_SYS_FUNC_EXIT	A function that's instrumented for profiling is exited
_NTO_TRACE_SYS_MAPNAME	*dlopen()* is called
_NTO_TRACE_SYS_MMAP	*mmap()* or *mmap64()* is called
_NTO_TRACE_SYS_MUNMAP	*munmap()* is called
_NTO_TRACE_SYS_PATHMGR	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_SLOG	A message is written to the system log

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:

This event:	Is emitted when a thread:
_NTO_TRACE_THCONDVAR	Enters the CONDVAR state
_NTO_TRACE_THCREATE	Is created
_NTO_TRACE_THDEAD	Enters the DEAD state
_NTO_TRACE_THDESTROY	Is destroyed

continued...

This event:	Is emitted when a thread:
_NTO_TRACE_THINTR	Enters the INTERRUPT state
_NTO_TRACE_THJOIN	Enters the JOIN state
_NTO_TRACE_THMUTEX	Enters the MUTEX state
_NTO_TRACE_THNANOSLEEP	Enters the NANOSLEEP state
_NTO_TRACE_THNET_REPLY	Enters the NET_REPLY state
_NTO_TRACE_THNET_SEND	Enters the NET_SEND state
_NTO_TRACE_THREADY	Enters the READY state
_NTO_TRACE_THRECEIVE	Enters the RECEIVE state
_NTO_TRACE_THREPLY	Enters the REPLY state
_NTO_TRACE_THRUNNING	Enters the RUNNING state
_NTO_TRACE_THSEM	Enters the SEM state
_NTO_TRACE_THSEND	Enters the SEND state
_NTO_TRACE_THSIGSUSPEND	Enters the SIGSUSPEND state
_NTO_TRACE_THSIGWAITINFO	Enters the SIGWAITINFO state
_NTO_TRACE_THSTACK	Enters the STACK state
_NTO_TRACE_THSTOPPED	Enters the STOPPED state
_NTO_TRACE_THWAITCTX	Enters the WAITCTX state
_NTO_TRACE_THWAITPAGE	Enters the WAITPAGE state
_NTO_TRACE_THWAITTHREAD	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 (e.g. AP_SCHED_BILL_AS_CRIT). For more information, 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 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.

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:

This event:	Is emitted when a virtual thread:
_NTO_TRACE_VTHCONDVAR	Enters the CONDVAR state
_NTO_TRACE_VTHCREATE	Is created
_NTO_TRACE_VTHDEAD	Enters the DEAD state
_NTO_TRACE_VTHDESTROY	Is destroyed
_NTO_TRACE_VTHINTR	Enters the INTERRUPT state
_NTO_TRACE_VTHJOIN	Enters the JOIN state
_NTO_TRACE_VTHMUTEX	Enters the MUTEX state
_NTO_TRACE_VTHNANOSLEEP	Enters the NANOSLEEP state

continued...

This event:	Is emitted when a virtual thread:
_NTO_TRACE_VTHNET_REPLY	Enters the NET_REPLY state
_NTO_TRACE_VTHNET_SEND	Enters the NET_SEND state
_NTO_TRACE_VTHREADY	Enters the READY state
_NTO_TRACE_VTHRECEIVE	Enters the RECEIVE state
_NTO_TRACE_VTHREPLY	Enters the REPLY state
_NTO_TRACE_VTHRUNNING	Enters the RUNNING state
_NTO_TRACE_VTHSEM	Enters the SEM state
_NTO_TRACE_VTHSEND	Enters the SEND state
_NTO_TRACE_VTHSIGSUSPEND	Enters the SIGSUSPEND state
_NTO_TRACE_VTHSIGWAITINFO	Enters the SIGWAITINFO state
_NTO_TRACE_VTHSTACK	Enters the STACK state
_NTO_TRACE_VTHSTOPPED	Enters the STOPPED state
_NTO_TRACE_VTHWAITCTX	Enters the WAITCTX state
_NTO_TRACE_VTHWAITPAGE	Enters the WAITPAGE state
_NTO_TRACE_VTHWAITTHREAD	Enters the WAITTHREAD state