QNX SOFTWARE SYSTEMS

QNX® Neutrino® Realtime OS: Kernel Benchmark Methodology

Table of Contents

Introduction

This document describes the outline and methodology of the QNX® Neutrino® kernel benchmarks suite, the results of which are available on a per-platform basis in the "QNX Neutrino Realtime OS: Kernel Benchmark Results" documents. It is recommended that this methodology overview be read in conjunction with those results to provide context as to what each individual test is measuring, the rationale behind each group of tests, and their relevance to real-world application performance. The following sections include an overview of the principles and operations involved in each test group, and details on any specific considerations and implementations.

Benchmark Methodology

The methodology for this benchmark series contains some assumptions regarding overhead. The average cost of a single instance of an operation is calculated from the total elapsed time required to perform a large number of iterations of each such operation. The number of iterations required to obtain a representative value is scaled to the clock speed of the host processor, typically in the order of millions. Averaging the benchmark in this fashion removes the need for a fine-granularity time-stamp and reduces variance. (For example, a single iteration may be skewed by an external or clock interrupt.)

The overhead of the empty loop itself is either insignificant or can be measured and eliminated. The average cost is considered a more appropriate measure than the worst-case time, as that metric can be adversely affected by such random factors as unrelated system activity and interrupt latencies, which are dependent on particular hardware peripherals and installed drivers. Elements that may legitimately affect the average performance of an operation, such as alternate kernel code paths or thread rescheduling and context switching, are accounted for with variants of each benchmark test designed to explicitly and consistently evoke those situations.

Kernel Entry

The QNX Neutrino realtime operating system architecture comprises a microkernel, the process manager, and extended services provided by user-level managers. The microkernel provides such core facilities as message passing, thread scheduling, timers, synchronization objects, and signals, whereas the process manager builds on the kernel facilities to provide additional process-level semantics, memory management, and pathname management. Optional extended services available include the file system, TCP/IP network protocols, and message queues.

The kernel is entered through a trap or software interrupt from a wrapper routine in the C library corresponding to each kernel primitive. Parameters and results are passed directly in registers or on the stack, and the kernel executes largely in the context of the calling thread, making it a very efficient interface. The process manager executes as a set of schedulable threads in its own process context, and parameters and results are exchanged using message passing. Extended facilities provided by external servers, which include named semaphores and message queues, require both message passing and two full context switches, which is why they incur the most overhead.

As a result, the relative performance of the various primitive operations profiled below is dependent on which layer implements that facility. This factor should also be considered when interpreting some external UNIX-derived benchmarks, which, for example, may use the `getppid()` call to illustrate the "system call overhead." Under QNX Neutrino this is an overly pessimistic measurement, as this particular routine is implemented by the process manager rather than by the microkernel. Since this is a low-bandwidth call by applications, it is not necessary to optimize this by implementing it within the kernel; rather the bias is towards keeping the microkernel small.

Kernel call

This test measures the overhead from a kernel call by timing the simplest possible facility provided by the microkernel; this is the `ClockId()` call, which returns a per-thread clock identifier for use in subsequent `ClockTime()` calls.

Benchmark Loop

```
ClockId(0, 0);
```

Process manager call

This test measures the overhead of making a process manager call by timing the simplest possible facility provided by the process manager; an example of this is the `getppid()` routine, which returns the parent of the calling process. Unlike the `getpid()` process identifier, which is invariant and therefore cacheable in thread local storage (TLS), the parent process is potentially changeable (consider a process whose parent terminates), meaning the process manager is required to implement this call.

Benchmark Loop

```
getppid();
```

Context Switching

Context switch operations are frequent in a realtime microkernel operating system. In such an environment, hardware interrupts, signals, message passing, and the manipulation of synchronization objects can trigger thread rescheduling, and non-core functionality is performed outside of the kernel by the process manager or external server processes. An efficient context switching implementation is essential to overall system performance.

Yield (self)

This test measures the overhead of a high-priority thread yielding the processor using the POSIX `sched_yield()` call. If threads of the same priority are ready to run, the calling thread is placed at the end of the ready queue for that priority, and a context switch occurs. In this case, as there are no other eligible threads, the running thread continues without a context switch.

Initialization

```
param.sched_priority = sched_get_priority_max(SCHED_FIFO);
sched_setscheduler(0, SCHED_FIFO, &param);
```

Benchmark Loop

```
sched_yield();
```

Yield (inter-thread, inter-process)

These tests measure the overhead of an inter-thread (or inter-process) context switch. The effect of the `sched_yield()` calls on two threads, which are created at the same high priority, is to alternate their use

of the processor and perform context switches between them. In a multiprocessor (SMP) environment, the "CPU affinity" of each thread is set to bind them to the same processor, ensuring that this test measures only the cost of context switching, without any inter-processor synchronization overhead. A full process context switch is typically more expensive than a thread context switch, as the kernel virtual memory subsystem has more overhead in switching the address space. (For example, a TLB flush is required on MMU architectures with untagged TLBs, such as the x86.)

Thread/Process 1 Benchmark Loop	*Thread/Process 2 Benchmark Loop*
`sched_yield();`	`sched_yield();`

Message Passing

Message-passing facilities are core primitives provided by the microkernel. Every QNX Neutrino application, including file systems, TCP/IP networking, and device drivers, is implemented as a team of cooperating threads and processes using the send/receive/reply messaging interface. Inter-process communication (IPC) occurs at specified transitions within the system, rather than asynchronously. The synchronous message-passing model facilitates robust client-server design, whereby application systems can be designed as a number of modular server processes handling client requests.

Messages

Message passing is inherently a blocking operation that synchronizes the execution of the sending thread; the act of sending the data also causes the sender to be blocked and the receiver to be scheduled for execution. This happens without requiring any explicit work by the microkernel to determine which thread or process to run next. Execution and data move directly from one context to another. Using message passing, a client sends a request to a server and becomes blocked. The server receives the messages in priority order from clients, processes them, and replies when it can satisfy a request. At this point the client is unblocked and continues.

Message MsgSend / MsgReceive / MsgReply

These tests measure the amount of time to perform a message pass between two different processes. Each cycle of the test loop requires three kernel calls and two context switches. A message containing a certain amount of data is sent in one direction with `MsgSend()`, and another amount of data is returned with `MsgReply()`. Four test configurations are used, whereby the size of the messages is varied as shown in the following table:

Test	MsgSend() bytes	MsgReply() bytes
No data	0	0
Send data	1024	0
Reply data	0	1024
Send + Reply data	1024	1024

Initialization

```
chid = ChannelCreate(0);
                                        coid = ConnectAttach(0, getppid(), chid,
                                                _NTO_SIDE_CHANNEL, 0);
```

Server Process Benchmark Loop

```
rcvid = MsgReceive(chid, msg1,
        len1, NULL);
MsgReply(rcvid, EOK, msg2, len2);
```

Client Process Benchmark Loop

```
MsgSend(coid, msg1, len1, msg2, len2);
```

Pulses

A pulse is a non-blocking, unidirectional message with a small data payload (four bytes). The pulse may be queued in the kernel if there is no blocked reader and it cannot be delivered immediately. Pulses are commonly used as a notification mechanism within interrupt handlers or for per-process timer expiry. For example, a pulse may be used as the associated event for `InterruptAttachEvent()` or `timer_create()`. Pulses may also be used for a server to signal to clients the occurrence of an event of interest, such as in `select()` processing.

Pulse MsgSendPulse / MsgReceivePulse

This test measures the time taken to send and receive a pulse within a single thread. Although each operation requires a kernel call, and the kernel must buffer the pulse, there are no context switches involved.

Initialization

```
chid = ChannelCreate(0);
coid = ConnectAttach(0, getpid(), chid, _NTO_SIDE_CHANNEL, 0);
```

Benchmark Loop

```
MsgSendPulse(coid, -1, 0, 0);
MsgReceivePulse(chid, &pulse, sizeof(pulse), NULL);
```

Pulse MsgSendPulse / MsgReceivePulse (inter-thread)

This test measures the time taken to send and receive a pulse using two threads of the same process. The thread sending the pulse is created at a lower priority than the one waiting to receive it. This relationship forces the threads to alternately block and unblock each other as the pulse is delivered without kernel buffering, and ensures a context switch is associated with each pulse transfer.

Initialization

```
chid = ChannelCreate(0);
coid = ConnectAttach(0, getpid(), chid, _NTO_SIDE_CHANNEL, 0);
```

Thread 1 Benchmark Loop | *Thread 2 Benchmark Loop (lower priority)*

```
MsgReceivePulse(chid, &pulse,     MsgSendPulse(coid, -1, 0, 0);
       sizeof(pulse), NULL);
```

Pulse MsgSendPulse / MsgReceivePulse (inter-process)

This test measures the time taken to send and receive a pulse between two different processes. The process sending the pulse is created at a lower priority than the one waiting to receive it. This relationship forces the processes to alternately block and unblock each other as the pulse is delivered without kernel buffering, and ensures a context switch is associated with each pulse transfer.

Initialization

```
chid = ChannelCreate(0);
                                    coid = ConnectAttach(0, getppid(), chid,
                                           _NTO_SIDE_CHANNEL, 0);
```

Process 1 Benchmark Loop | *Process 2 Benchmark Loop (lower priority)*

```
MsgReceivePulse(chid, &pulse,        MsgSendPulse(coid, -1, 0, 0);
       sizeof(pulse), NULL);
```

Synchronization

QNX Neutrino supports a full complement of POSIX 1003.1 thread synchronization primitives, including mutexes, semaphores, and condition variables. These facilities allow multiple threads, of the same process or of different processes, to protect critical sections of code or to coordinate access to a shared resource or memory area.

Mutexes

A mutex is an appropriate simple synchronization object for use in permitting only a single thread at a time into a critical section of code. For example, when updating a shared doubly linked list, if multiple threads were allowed to enqueue/dequeue elements in an uncontrolled manner, the link pointers could become inconsistent and corrupt the list. By controlling access to the list using a mutex only, a single thread will be able to modify it at any one time, and any other thread attempting to do so will block until the mutex is unlocked. A mutex is a more elegant, fine-grained, and multiprocessor-safe mechanism for performing such synchronization than disabling interrupts or thread switching. An efficient mutex implementation is thus important to the performance of all multithreaded applications. In fact, many internal services of QNX Neutrino, such as the file system, network protocol stacks, and device drivers, as well as the implementation of thread safety where required within the C library, rely heavily on mutexes to provide such synchronization.

Uncontested pthread_mutex_lock / pthread_mutex_unlock

This test measures the time taken to lock and unlock a mutex. Because the normal use of a mutex is to enclose a small critical section of code within the scope of the locked mutex, it is appropriate to measure this complete cycle as a combined item. The mutex is created with default attributes and will be

uncontested in this test because no other thread is attempting to acquire a lock. Therefore, on most platforms, it will be unnecessary to enter the kernel, as the mutex can be safely manipulated using native processor support (with instructions designed for atomic memory modification, such as the x86 `cmpxchg` or the `lwarx`/`stwcx` sequence of the PowerPC). On processors without this functionality, the mutex operations must either enter the kernel or emulate a suitable synchronization primitive.

Initialization

```
pthread_mutex_init(&mutex, NULL);
```

Benchmark Loop

```
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
```

Unavailable pthread_mutex_trylock

This test measures the time taken to attempt to lock an already locked mutex. The mutex is created with default attributes. As this is an unsuccessful non-blocking probe of the mutex state, it is again unnecessary to enter the kernel or to reschedule any threads.

Initialization

```
pthread_mutex_init(&mutex, NULL);
pthread_mutex_lock(&mutex);
```

Benchmark Loop

```
pthread_mutex_trylock(&mutex, NULL);
```

In-kernel pthread_mutex_lock / pthread_mutex_unlock

This test measures the time to lock and unlock a mutex by entering the kernel. Unlike the previous tests, which can exploit native processor support to directly manipulate the mutex, this set of operations is forced into the kernel. The PTHREAD_PRIO_PROTECT attribute is used to achieve this, as locking such a mutex implies the kernel may make an immediate priority change. In comparison, the default PTHREAD_PRIO_INHERIT attribute adjusts thread priorities to avoid priority inversion only when a higher-priority thread contests the locked mutex. Although this test forces kernel entry, the mutex is uncontested, and thus no rescheduling or context switches are required.

Initialization

```
pthread_mutexattr_init(&attr);
pthread_mutexattr_setprotocol(&attr, PTRHEAD_PRIO_PROTECT);
pthread_mutexattr_setprioceiling(&attr, getprio(0));
pthread_mutex_init(&mutex, &attr);
```

Benchmark Loop

```
pthread_mutex_lock(&mutex);
pthread_mutex_unlock(&mutex);
```

Contested pthread_mutex_lock / pthread_mutex_unlock

This test measures the time to lock and unlock a contested mutex (that is, one locked or blocked-on by another thread). During this test two threads access four mutexes in a lock-step fashion to force such locking collisions. Entry into the kernel is necessary to resolve this contention and to perform the required thread rescheduling and context switching as the threads alternately block and unblock each other. Each cycle of the test loops comprises four contested mutex lock/unlock pairs and four uncontested mutex operations. These additional operations, which are necessary to interlock the contested accesses and ensure collisions, will be handled without kernel intervention and their overhead can be measured and accounted for in the final result.

Initialization

```
pthread_mutex_init(&mutex1);
pthread_mutex_init(&mutex2);
pthread_mutex_init(&mutex3);
pthread_mutex_init(&mutex4);
pthread_mutex_lock(&mutex4);      pthread_mutex_lock(&mutex1);
```

Thread 1 Benchmark Loop	*Thread 2 Benchmark Loop*
`pthread_mutex_lock(&mutex1);`	`pthread_mutex_lock(&mutex3);`
`pthread_mutex_lock(&mutex3);`	`pthread_mutex_lock(&mutex2);`
`pthread_mutex_unlock(&mutex1);`	`pthread_mutex_unlock(&mutex3);`
`pthread_mutex_unlock(&mutex4);`	`pthread_mutex_unlock(&mutex1);`
`pthread_mutex_lock(&mutex2);`	`pthread_mutex_lock(&mutex4);`
`pthread_mutex_lock(&mutex4);`	`pthread_mutex_lock(&mutex1);`
`pthread_mutex_unlock(&mutex2);`	`pthread_mutex_unlock(&mutex4);`
`pthread_mutex_unlock(&mutex3);`	`pthread_mutex_unlock(&mutex2);`

Semaphores

A semaphore is a flexible synchronization object that has a non-negative integer count and a set of blocked threads associated with it. A semaphore can be used to control access to a pool of resources or to indicate the occurrence of events as in a producer/consumer paradigm. Semaphores are explicitly defined to work between processes and with signals, which makes them a common method of inter-process or signal-handler synchronization in portable code. It is typical to use a mutex to synchronize between threads in the same process, and to use a semaphore to synchronize between different processes. Unlike the QNX Neutrino implementation of a mutex, which optimizes the uncontested cases, a semaphore operation always results in kernel entry.

Unnamed semaphore unavailable sem_trywait

This test measures the time taken to attempt to wait on a zero-valued (locked) semaphore. Although each semaphore operation requires a kernel call, there are no context switches involved since this is a non-blocking unsuccessful probe of the semaphore state.

Initialization

```
sem_init(&sem, 0, 0);
```

Benchmark Loop

```
sem_trywait(&sem);
```

Unnamed semaphore sem_post / sem_wait (self)

This test measures the time taken to increment and decrement a semaphore within a single thread. Although each semaphore operation requires a kernel call, there are no context switches involved.

Initialization

```
sem_init(&sem, 0, 0);
```

Benchmark Loop

```
sem_post(&sem);
sem_wait(&sem);
```

Unnamed semaphore sem_post / sem_wait (inter-thread)

This test measures the time taken to increment and decrement a semaphore using two threads of the same process. The thread incrementing the semaphore is created at a lower priority than the one attempting to decrement it. This relationship forces the threads to alternately block and unblock each other as the semaphore count toggles between 0 and 1, and ensures a context switch is associated with each semaphore access.

Initialization

```
sem_init(&sem, 0, 0);
```

Thread 1 Benchmark Loop	*Thread 2 Benchmark Loop (lower priority)*
`sem_wait(&sem);`	`sem_post(&sem);`

Unnamed semaphore sem_post / sem_wait (inter-process)

This test measures the time taken to increment and decrement a semaphore between two different processes. Because the unnamed semaphore is to be shared among processes, the semaphore object is created in a shared-memory region in order to be accessible by both processes. The process incrementing the semaphore is created at a lower priority than the one attempting to decrement it. This relationship forces the processes to alternately block and unblock each other as the semaphore count toggles between 0 and 1, and ensures a context switch is associated with each semaphore access.

Initialization

```
fd = shm_open("/sem", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR);
ftruncate(fd, sizeof(sem_t));
sem = mmap(NULL, sizeof(sem_t), PROT_READ|PROT_WRITE, MAP_SHARED, fd, 0);
sem_init(sem, !0, 0);
```

Process 1 Benchmark Loop	*Process 2 Benchmark Loop (lower priority)*
`sem_wait(sem);`	`sem_post(sem);`

Named semaphore sem_trywait / sem_post / sem_wait

These tests measure the time taken to increment and decrement a named semaphore. A named semaphore is a mechanism for processes that are not sharing a common address space to synchronize with one another. Named semaphores in QNX Neutrino are an extension provided by an external server (`mqueue`), rather than being primitive kernel objects, and as such offer lower performance than their unnamed counterpart. These tests are similar to those above, with the sole difference being in the way the named semaphore is created.

Initialization

```
sem = sem_open("/sem", O_ACCMODE|O_CREAT, S_IRUSR|S_IWUSR, 0);
```

Timers

QNX Neutrino implements POSIX 1003.1 clock and timer facilities, including per-process notification of timer expiry via realtime signals (and other event types, including native pulses). Timers are primitives provided directly by the microkernel, and hence may be manipulated very efficiently.

Timer timer_create / timer_delete

This test measures the time taken to create and destroy a per-process timer. An explicit user event, in this case a `SIGALRM` signal, is associated with the timer.

Initialization

```
SIGEV_SIGNAL_INIT(&event, SIGALRM);
```

Benchmark Loop

```
timer_create(CLOCK_REALTIME, &event, &timer);
timer_delete(timer);
```

Timer arming timer_settime

This test measures the time taken to reset and re-arm a per-process timer. In the general POSIX interface, the expiry time may be specified as either a relative or absolute value, and the time remaining from a previous arming of the timer may be returned. In this case, an absolute future time is used (so that the timer will not fire during the benchmark) and the previous timeout details are not requested.

Initialization

```
timer_create(CLOCK_REALTIME, NULL, &timer);
timeout.it_value.tv_sec = time(NULL) + 60, timeout.it_value.tv_nsec = 0;
timeout.it_interval.tv_sec = timeout.it_interval.tv_nsec = 0;
```

Benchmark Loop

```
timer_settime(timer, TIMER_ABSTIME, &timeout, NULL);
```

Signals

Signals are used by the operating system to report synchronous exceptions (such as `SIGSEGV`, `SIGBUS`, or `SIGFPE`), and by application processes as asynchronous event notifications (via a `sigevent` structure or an explicit `kill()`). QNX Neutrino implements the Realtime Extensions to the POSIX 1003.1 signal facilities. These extensions include the `SA_SIGINFO` attribute, the association of an application-defined value with each signal, the ability to queue pending signals, and the prioritized receipt of signals within the defined realtime range.

Signal delivery (self)

This test measures the time taken to raise and handle a signal within an individual thread. An empty signal-catching function is installed for execution upon receipt of the signal. Although each signal operation requires a kernel call, there are no context switches involved.

Initialization

```
void emptyhandler(int signo) {}

sa.sa_handler = emptyhandler;
sigfillset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGUSR1, &sa, NULL);
```

Benchmark Loop

```
kill(pid, SIGUSR1);
```

Signal delivery (inter-thread, inter-process)

This test measures the time taken to generate and deliver a signal between two threads of the same process or between two different processes. The entity generating the signal is created at a lower priority than the one accepting it. This relationship ensures that the delivery of each signal requires a context switch and that pending signals are neither queued nor discarded, as the signal will be immediately accepted.

Initialization

```
void emptyhandler(int signo) {}

sa.sa_handler = emptyhandler;
sigfillset(&sa.sa_mask);
sa.sa_flags = 0;
sigaction(SIGUSR1, &sa, NULL);
sigfillset(&iset), sigdelset(&iset, SIGUSR1);
```

Thread/Process 1 Benchmark Loop

```
SignalWaitinfo(&iset, NULL);
```

Thread/Process 2 Benchmark Loop (lower priority)

```
kill(pid, SIGUSR1);
```

Threads

A thread is a single flow of control within a process, where a process is an address space in which one or more threads execute. Threads provide a framework for certain classes of application, in particular client/server models, to leverage concurrency and exploit any underlying SMP parallelism. The QNX Neutrino microkernel directly supports POSIX threads as schedulable entities, although the process manager is used to provide additional process-level semantics.

Thread pthread_create / pthread_join

This test measures the time to create, schedule, and terminate a thread. The thread executes an empty function, the return from which is an implicit call to `pthread_exit()`.

Initialization

```
void *emptyfunction(void *arg) {return(arg);}

pthread_attr_init(&attr);
pthread_attr_setstacksize(&attr, PTHREAD_STACK_MIN);
```

Benchmark Loop

```
pthread_create(&tid, &attr, emptyfunction, NULL);
pthread_join(tid, NULL);
```

Thread pthread_create / pthread_join (non-lazy stack)

This test is similar to the above thread creation benchmark, with the exception that the thread stack attribute is marked as `PTHREAD_STACK_NOTLAZY`, rather than the default on-demand "lazy stack" policy. This variant causes the up-front physical allocation of the stack during thread creation, avoiding a virtual memory (VM) page fault when the new thread first runs.

Initialization

```
pthread_attr_setstacklazy(&attr, PTHREAD_STACK_NOTLAZY);
```

Thread pthread_create / pthread_join (user stack)

This test is similar to the above thread creation benchmark, with the exception that an explicit block of memory is provided as the stack for the new thread. This variant does not require the kernel to allocate any memory or the VM subsystem to adjust the address space of the process.

Initialization

```
char userstack[PTHREAD_STACK_MIN + sizeof(struct _thread_local_storage)];

pthread_attr_setstackaddr(&attr, userstack);
```

Message Queues

POSIX defines a set of message-passing facilities known as message queues, which allow for the transfer of arbitrary data between cooperating processes. Under QNX Neutrino, message queues are not a core kernel primitive, but are implemented through an optional user-level manager called `mqueue`. Although aspects of the interface appear to be asynchronous, it is built on top of QNX Neutrino native synchronous message passing, using the `mqueue` server to broker the transaction and perform store-and-forward buffering. Message queues thus offer lower absolute performance than direct message passing, but are useful in specific situations. For example, they provide flexibility with their buffering and asynchronous notification facilities, and are a portable IPC mechanism for application code that must run under multiple operating systems.

Unavailable mq_receive

This test measures the time taken to perform a non-blocking read attempt from an empty message queue. Although no user data is transferred, this operation involves context switches and message passing with the `mqueue` server to determine the queue state.

Initialization

```
mq = mq_open("/mq", O_RDWR|O_NONBLOCK|O_CREAT, S_IRUSR|S_IWUSR, NULL);
```

Benchmark Loop

```
mq_receive(mq, NULL, 0, NULL);
```

Empty mq_send / mq_receive (self)

This test measures the time for a single thread to alternately write a 256-byte message to an empty message queue and then read back that same message. Because a single thread is manipulating the message queue, there is no blocked reader for the message when it is sent, so it must be explicitly stored within `mqueue`. Since the message queue is empty, the priority of the message is irrelevant, so it can be placed at the front of the queue. This test requires two sets of context switches and message passing with the `mqueue` server, plus implementation overheads due to message buffering.

Initialization

```
attr.mq_msgsize = 256, attr.mq_maxmsg = 1;
mq = mq_open("/mq", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR, &attr);
```

Benchmark Loop

```
mq_send(mq, msg, 256, 0);
mq_receive(mq, msg, 256, NULL);
```

Empty mq_send / mq_receive (inter-thread, inter-process)

These tests measure the time to transfer a 256-byte message between two threads (or two different processes) via an intermediary empty message queue. The entity sending the message is created at a lower priority than the one receiving it. This relationship forces the threads (or processes) to alternately block

and unblock each other as the messages are transferred through the message queue. This ensures the empty message queue will have a blocked reader, enabling an optimization of relaying the messages rather than buffering them within `mqueue`. This test requires two sets of context switches and message passing with the `mqueue` server.

Initialization

```
attr.mq_msgsize = 256, attr.mq_maxmsg = 1;
mq = mq_open("/mq", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR, &attr);
```

Thread/Process 1 Benchmark Loop	*Thread/Process 2 Benchmark Loop (lower priority)*
`mq_receive(mq, msg, 256, NULL);`	`mq_send(mq, msg, 256, 0);`

Full mq_send / mq_receive

This test measures the time for a single thread to alternately read a 256-byte message from a full message queue and then write back another message. Because a single thread is manipulating the message queue, there is no blocked reader for the message when it is sent, requiring it to be explicitly stored within `mqueue`. Since the message queue is non-empty, the priority of each message must be used to insert it into its correct sorted position within the queue. The capacity of the message queue is 64 messages, and all messages written into the queue are assigned random priorities. This test requires two sets of context switches and message passing with the `mqueue` server, plus implementation overheads due to message buffering and sorting.

Initialization

```
attr.mq_msgsize = 256, attr.mq_maxmsg = 64;
mq = mq_open("/mq", O_RDWR|O_CREAT, S_IRUSR|S_IWUSR, &attr);
for (fill = 0; fill < 64; ++fill)
       mq_send(mq, msg, 256, rand() % MQ_PRIO_MAX);
```

Benchmark Loop

```
mq_receive(mq, msg, 256, &priority);
mq_send(mq, msg, 256, rand() % MQ_PRIO_MAX);
```

Summary

The methodology described in this document outlines the rationale behind the kernel benchmark tests prepared by QNX Software Systems. Actual benchmark results for the tests described in the above sections are available on a per-platform basis in the "QNX Neutrino Realtime OS: Kernel Benchmark Results" documents. Please contact your local QNX sales representative for more information.

About QNX Software Systems

Founded in 1980, QNX Software Systems is the industry leader in realtime, microkernel OS technology. The inherent reliability, scalable architecture, and proven performance of the QNX Neutrino RTOS make it the most trusted foundation for future-ready applications in the networking, automotive, medical, and industrial automation markets. Companies worldwide like Cisco, Ford, Johnson Controls, Siemens, and Texaco depend on the QNX technology for their mission- and life-critical applications. Headquartered in Ottawa, Canada, QNX Software Systems maintains offices in North America, Europe, and Asia, and distributes its products in more than 100 countries worldwide.

www.qnx.com

www.qnx.com