expresslogic

Proper Priority Assignment Can Have a Major Effect on Real-Time Performance

Adjusting thread priorities can reduce context switching and increase overall throughput by 80%

Real-time software applications consist of multiple tasks or threads, each performing a portion of the system's workload under the management of a real-time operating system (RTOS). In a real-time system, a thread's priority reflects the urgency of its work, relative to other threads and activities of the system. RTOSes strive at all times to run the most urgent work that needs to be performed. A priority-based, preemptive scheduling RTOS will always run the highest priority thread that is "READY" (not blocked or suspended) to run. But, the ultimate goal of many real-time systems is throughput – the more packets processed, bytes streamed, frames rendered per second, the better. Sometimes, unnecessary real-time responsiveness can actually add overhead, and reduce the system's performance. In those instances, there is a tradeoff between responsiveness and throughput. One example of such a tradeoff occurs when a system uses high priorities in an attempt to achieve fast response, but suffers increased overhead and reduced throughput as a result. The assignment of higher priorities to those application threads that respond to actions of other threads, or to events that occur outside the system, is effective in minimizing latency of such threads, and might be an appropriate for most applications. However, many times the system can achieve greater throughput by using a different approach to priority assignment that does not require each thread to have a high priority and the fastest possible response.

By assigning unique priorities to all threads, a strict hierarchy of urgency is established. Since a preemptive scheduling RTOS will always run the highest priority (most urgent) thread that is "READY," this assures the intended level of responsiveness for each thread. In an alternate approach, the RTOS runs threads of equal priority in a Round-Robin fashion, allowing each one to complete its work, or voluntarily relinquish its use of the CPU, before moving on to the next. In many cases, overall system throughput can be higher with the round-robin approach, and so it might be preferred if throughput is the more important system objective. In this paper, we will examine just how priority assignment affects system performance, and show how the use of unique priorities can actually add overhead and reduce overall throughput.

The Context-Switch

A context-switch is a complex procedure in which the RTOS saves all the information being used by the running thread (its "context") and loads the context of another thread in its place. A thread's context includes its working register set, program counter, and other thread-critical information. The context is saved on the thread's stack, or in a thread control block data structure, from which it gets re-loaded when the RTOS wants to run that thread again.

Context switches generally are the single most time-consuming RTOS operation in a real-time system, often taking hundreds of cycles to execute. The amount of processing varies from RTOS to RTOS, and processor architecture to architecture, but generally involves the following operations:

Step	Operation	Cycles
1	Save the current thread's context (ie: GP and FPU register values and PC) on the stack.	20 - 100
2	Save the current stack pointer in the thread's control block.	2 - 20
3	Switch to the system stack pointer.	2 - 20
4	Return to the scheduler.	2 - 20
5	Find the highest priority thread that is ready to run.	2 - 50
6	Switch to the new thread's stack.	2 - 50
7	Recover the new thread's context.	20 - 100
8	Return to the new thread at its previous PC.	2 - 40
9	Other processing	0 - 100
	TOTAL CYCLES	**50 - 500**

Figure 2 – A typical context switch involves a number of operations, each one requiring a number of CPU cycles

Because of all the processing it requires, a context switch is one of the most important measures of real-time performance in embedded systems. While all RTOSes and processors go to great lengths to optimize the performance of their context switch operation, the application developer must ensure that a system performs as *few* context switches as possible. For a given application, the way priorities are assigned to individual threads can have a significant impact on the number of context switches performed. In particular, by running multiple threads at the same priority, rather than assigning them each a unique priority, the system designer can avoid unnecessary context switches and reduce RTOS overhead. This results in greater throughput, often the ultimate goal of the system.

Other Benefits

Assigning multiple threads the same priority also makes it possible for the RTOS to properly implement priority inheritance, round-robin scheduling, and time-slicing. Each of these mechanisms is important in a real-time system, and is difficult, if not impossible to implement, without running multiple threads at the same priority. Each can be used to keep system overhead low and—perhaps more importantly—to keep system behavior understandable. Express Logic's ThreadX RTOS provides developers with a choice of priority assignment

approaches, either unique priorities for each thread, or multiple threads at the same priority, or a combination of both. In this paper, we use ThreadX to implement each strategy on the same application code, and compare the results. To analyze system behavior in each case, we use our TraceX graphical analysis tool to illustrate, and to actually measure the difference in overhead and overall system performance. The result will be to show that, for certain types of real-time systems, the use of unique priorities can reduce overall system throughput, due to the higher number of context switch operations that result from that approach.

An Example System

Message passing is a common element of many real-time systems. A simple, but common scenario is for a thread to be "waiting" for a message to appear in a message queue, and when the message appears, the waiting thread becomes "READY." The RTOS is responsible for keeping track of which threads are READY, which are waiting, and for recognizing when an event enables a waiting thread to become READY. Our example system will use this scenario to see how each approach to priority assignment handles the resulting activity and throughput.

Preemption

When a thread with priority higher than the active thread becomes READY to run (e.g., because the message it was waiting for finally arrived), the RTOS preempts the active thread. This preemption results in a context switch in which the context of the active thread is saved, the context of the higher priority thread is loaded, and the higher priority thread then runs on the CPU. This enables the higher priority thread to process the message immediately – which results in greater responsiveness. But, context switches are fairly expensive in terms of processor cycles used, so the benefit to responsiveness has to be weighed against the additional overhead that is introduced. Preemption operates differently between a system of threads with unique priorities, as opposed to a system of threads all at the same priority. As a result, context switching occurs differently in each type of system.

To illustrate the effect that various methods of priority assignment can have on context switching, consider a system with four threads, named A, B, C, and D. In this example, the threads operate in a producer-consumer fashion, with Thread D the producer thread, sending a message to each of three queues: A, B, and C. Thread D performs this in a loop 3 times, for a total of nine (9) messages sent.

Threads A, B, and C each check for a message from Thread D in one of the 3 queues. If one is found, they retrieve it. If not, they wait until one arrives. After retrieving a message from the queue, the threads loop to look for another, in a while(1) infinite loop. The threads "suspend" (i.e., are no longer ready to run) if no message is available in the queue, and "resume" when a message appears (sent from thread D). Each sequence of Threads A, B, C, and D (nine messages sent and nine messages retrieved) will constitute a "Cycle" of this example application, as shown below in Figure-3:

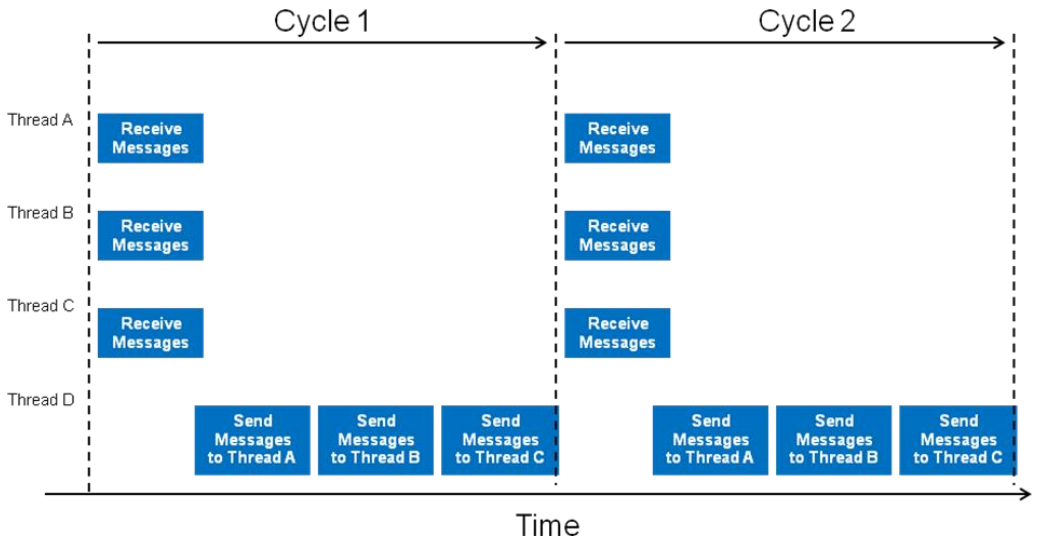

Figure 3 – Operational flow of example program, showing "cycles" consisting of Thread D sending messages to Threads A, B, and C and Threads A, B, and C

The operational code for this simple system would look something like the following:

```
 void thread_a_entry()
{
        while(1)
        {
                receive_message_from_thread_a_queue();
        }
}
void thread_b_entry()
{
        while(1)
        {
                receive_message_from_thread_b_queue();
        }
}
void thread_c_entry()
{
        while(1)
        {
                receive_message_from_thread_c_queue();
                        }
}
void thread_d_entry()
{
        while(1)
        {
                For 1-3
                send_message_to_thread_a_queue();
                send_message_to_thread_b_queue();
                send_message_to_thread_c_queue();
                End Loop
        }
        Relinquish;
}
```

Figure 4 – Pseudo code for example program. Note that Threads A, B, and C continuously loop retrieving messages from Thread D. Thread D sends a number of messages to each of the other threads, then relinquishes.

Two Cases

To illustrate how the priorities of the threads impact the total number of context switches performed by this system, we'll examine two cases. In "Case-1", all threads are assigned the same priority, and will execute in a round-robin fashion. In Case-2, the four threads are assigned unique priorities. In each case, we'll measure the number of context switches performed, as well as the time it takes to complete an equal amount of work. Figure-5 illustrates the priority assignments in each case.

In Case-1, where all threads have the same priority, the threads run in a round-robin fashion where Threads A. B, and C each run until blocked waiting for another message to appear in its queue. Thread D sends nine messages, then relinquishes its turn so the other threads can run. In Case-2, the highest priority thread that is ready to run always runs. This means that Thread A will always run when a message is in its queue. If not, then Thread B will run if a message is in its queue, and so on for Thread C. Eventually, all 3 queues will be empty, enabling Thread D to run and send more messages. To see how priority assignment affects performance, we will use TraceX to measure the time it takes to send/receive all nine messages in each case. First, though, we will examine the event trace to see what events occur in each case. We will see that there is a significant difference between the two cases in the number of context switches performed, and this will result in a significant difference in performance between the two cases.

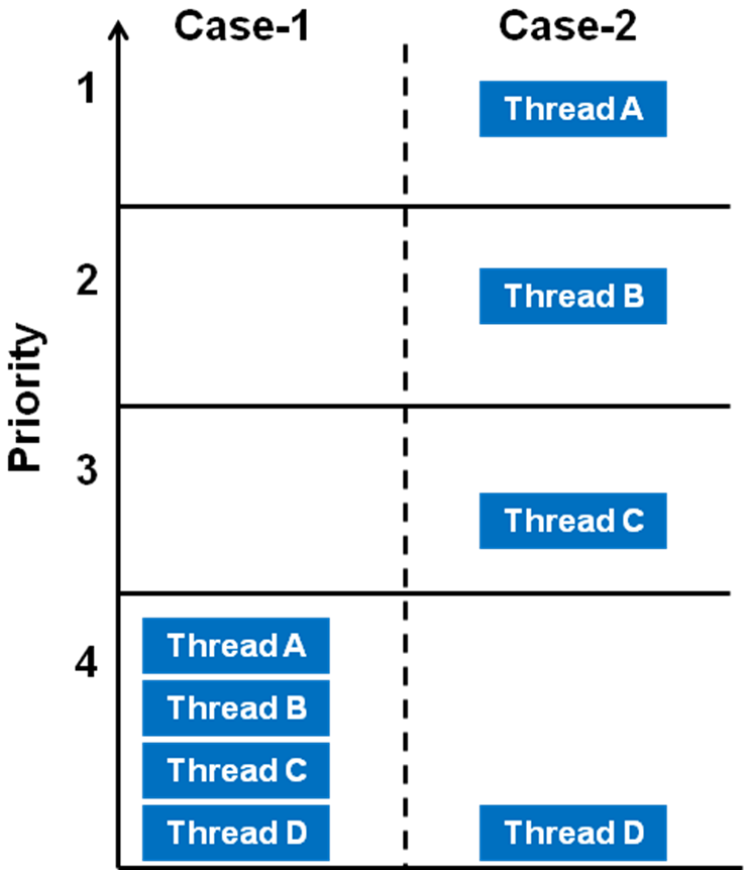

Figure 5 –To measure the impact of priority assignment, we set up two cases—one where all threads have the same priority and one where they each have a unique priority.

Case-1: All threads have the same priority (A=4, B=4, C=4, D=4)
In Case-1, we've assigned each thread a priority of 4. When all threads have the same priority, they will run in a round-robin fashion, in the order of their creation (A, B, C, D). In Figure 6, an execution event trace shows how the example operates, once Thread D has populated the message queues. Note how Threads A, B, and C each retrieve three messages from their respective queues, then suspend waiting for more. When all three have run, Thread D once again runs, populating each queue with three more messages. This sequence continues:

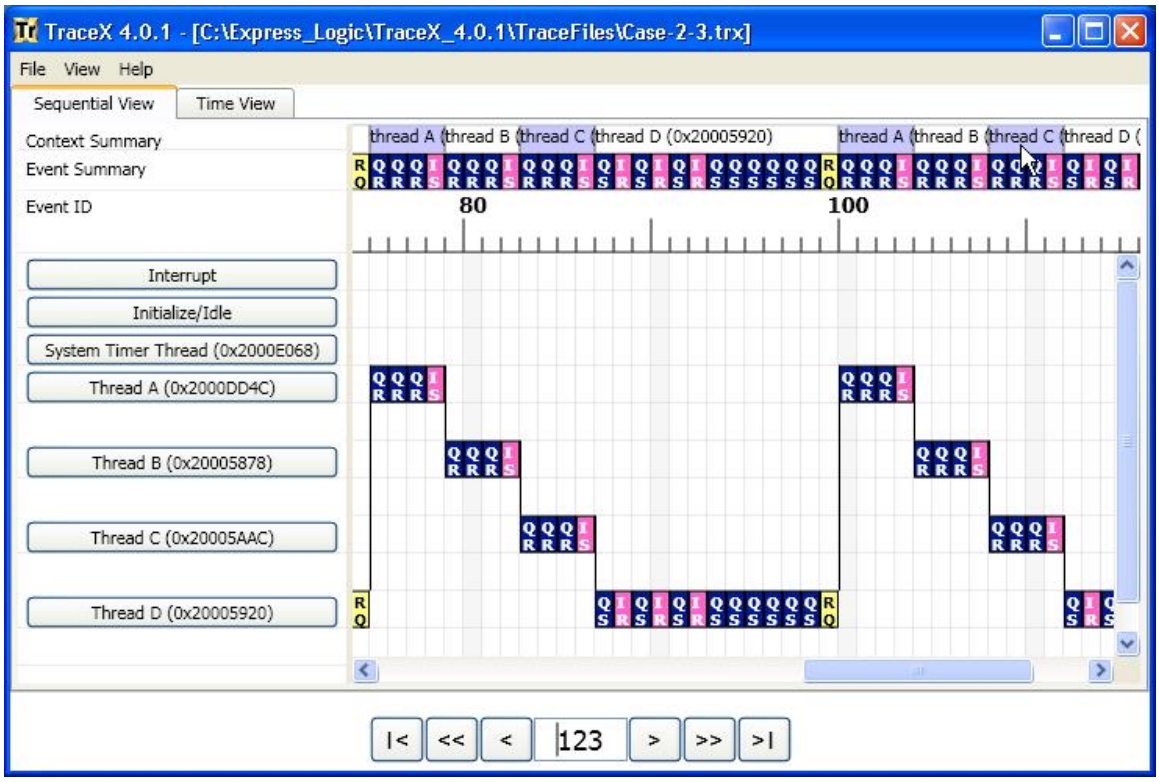

Figure 6 – Event trace of Case-1: Equal Priorities. This shows each of Threads A, B, and C retrieving three messages, then Thread D sending each Thread three more messages. This cycle is repeated continuously.

Threads A, B, and C each read 3 messages from their message queues. The "QR" (queue read) event indicates a successful read of one message from the queue. After 3 messages are read, the queue is empty, and the thread suspends, waiting for Thread D to send more messages. The "IS" (internal suspend) event indicates that the RTOS suspends the thread and returns to its scheduler which initiates a context switch to the next thread in the round-robin sequence. Thread D, the "producer" thread, eventually gets to run, and sends more messages to the queues (as shown by the "QS" event). Note that as each of the first three messages is sent by Thread D (as indicated by the "QS" event), there is an Internal Resume ("IR") event. This IR event refers to the fact that the thread waiting for that message becomes "ready" to run, but must wait its turn in the round-robin sequence. After the third message is sent (one to each waiting thread), the subsequent messages do not cause another IR, since those threads already have been resumed (made "ready") by the first message, which has not yet been retrieved.

In this case, there is exactly one context switch each time a thread completes its processing (is suspended waiting for a message to be put on its queue, or is finished sending messages in the case of Thread D), allowing the next thread in turn to run. The result is that there are a total of four context switches per "cycle" between Thread A's retrieval of the first message sent to it by Thread D. Note below in Figure 6, the Start and Stop boundaries of one application cycle. Context switches between the Start and Stop are numbered, and the total (4) is reflected in the "performance Statistics" display superimposed in the upper-right of Figure-7.

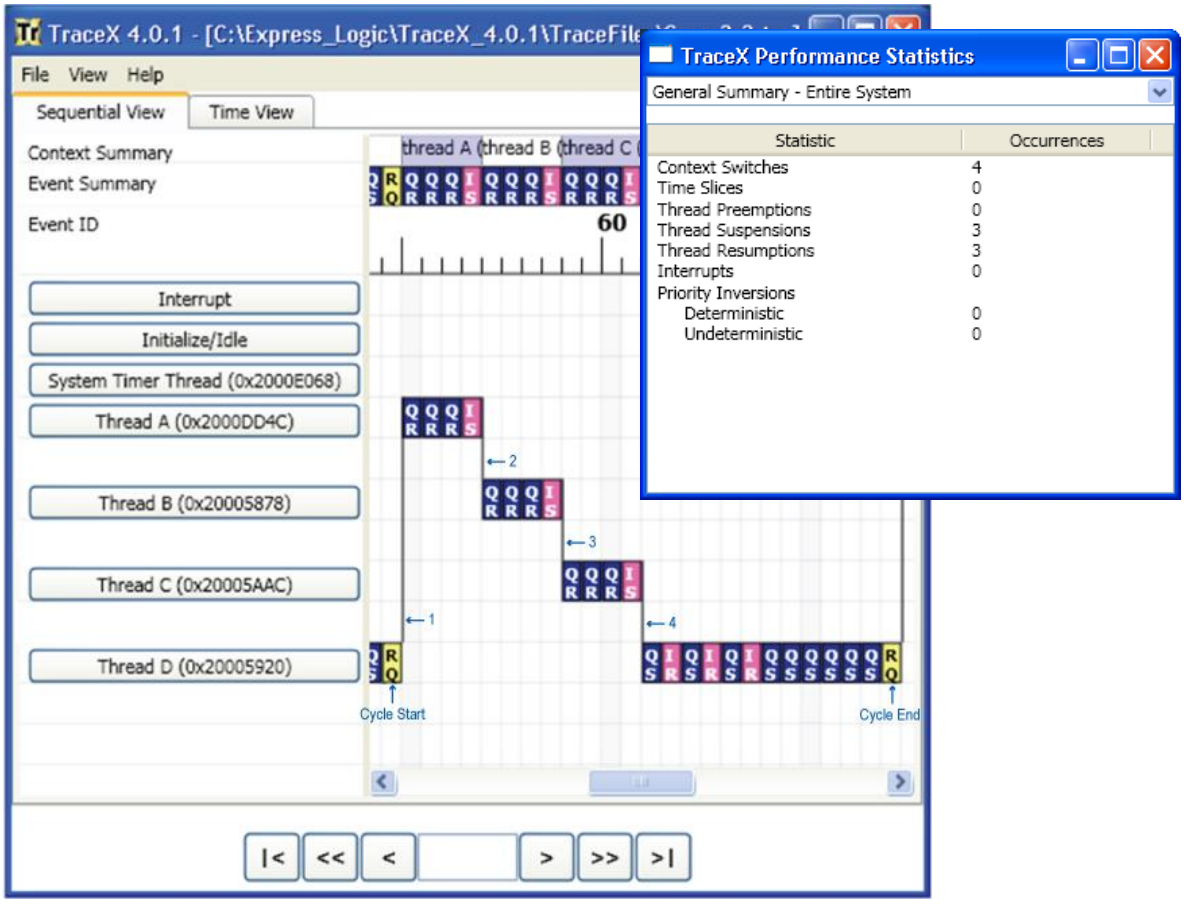

Figure 7 – Context Switch count for Case-1. Notice that only four context switches are required for a complete cycle of nine messages sent and received. The insert shows the count of various RTOS operations.

In Case-1, nine messages are sent, nine messages are received, and four context switches occur.

Case-2: All threads are given unique priorities

In Case-2, each thread is assigned a unique priority. Thread A=1, Thread B=2, Thread C=3, and Thread D=4. Since all threads have unique priorities, the highest priority thread that is ready to run is the one that the RTOS will run. Figure 8 below shows the event sequence for Case-2:

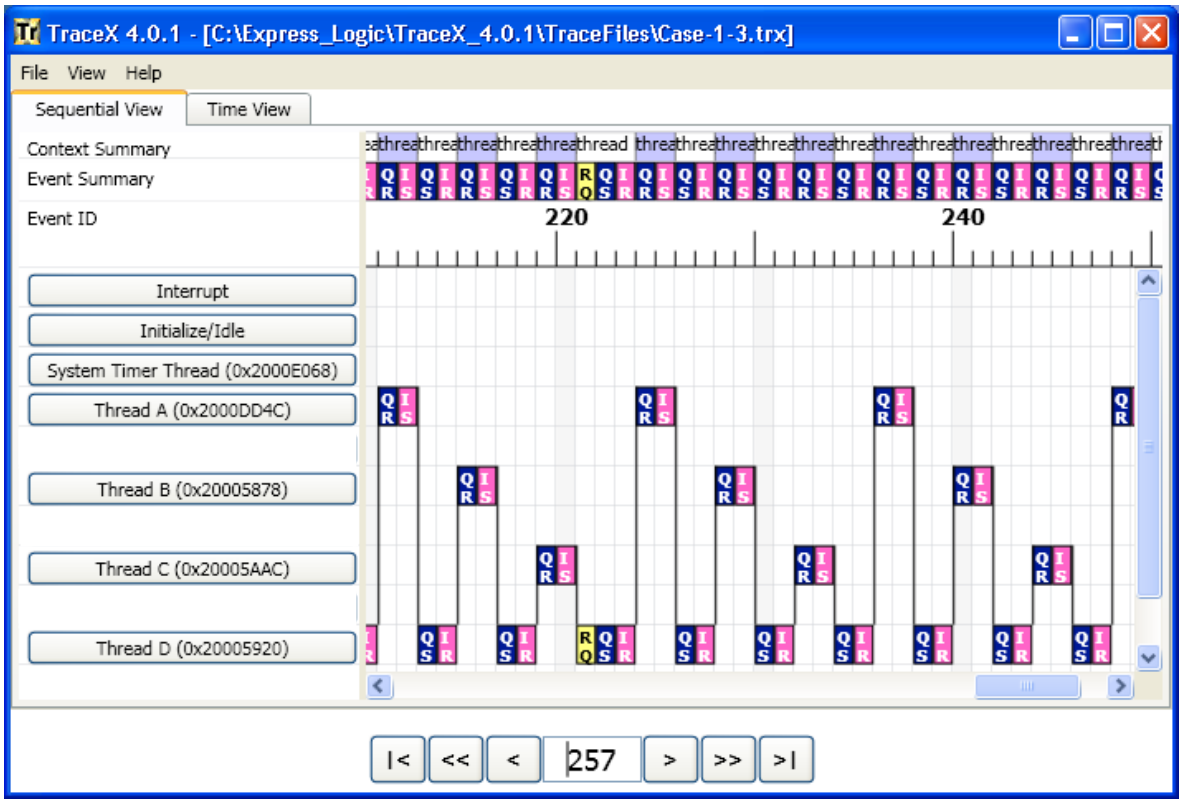

Figure 8 – Processing flow in Case-2. Note that after each message is sent by Thread D in this case, a preemption occurs and control is immediately transferred to the thread waiting for that message.

Figure 8 above shows a period of processing that occurs repeatedly, immediately following Thread D's sending a message to Thread A. Here, as we can see, when a message is sent by Thread D, it causes an immediate Internal Resume ("IR" event), just as in Case-1. But, unlike Case-1, because all of the threads waiting for a message in Case-2 have a higher priority than Thread D, the receiving thread becomes the highest priority thread that is ready to run. As a result, the RTOS preempts Thread D, and performs a context switch to that thread (Context Switches #2, 4, 6, 8, 10, 12, 14, 16, and 18 shown in Figure 9 below). This is different from the events of Case-1, where the waiting threads all had the same priority as Thread D, so no preemption was caused as Thread D sent its messages to the queues. Here, once each thread retrieves its message from the queue ("QR" event), it once again goes into a suspended state, waiting for another message (as indicated by the Internal Suspend, or "IS" event), and the RTOS does another context switch.

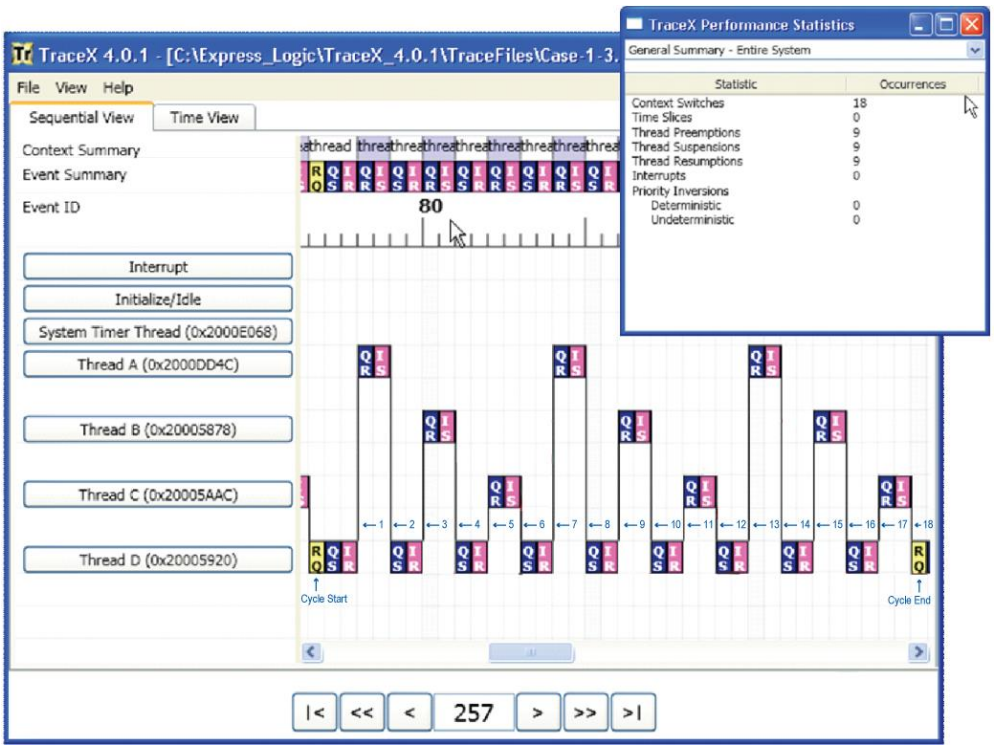

Figure 9 – Context Switches for a complete Cycle in Case-2. Note additional RTOS event counts shown in the insert.

In Case-2, nine messages are sent, and nine are retrieved, exactly the same as in Case-1. But, in Case-2, eighteen context switches occur for the same nine messages, as opposed to four in Case-1, an increase of 350%. This can be seen between the two "QR" events for Thread A noted above in Figure 9, compared to the previous case where only four context switches occur (Figure 7). The unique priority assignment strategy of Case-2, where the threads that become ready are higher ion priority than the thread making them ready, results in a significantly greater number of context switches compared to the assignment of the same priority to the threads, as used in Case-1. Through the selection of thread priorities, the exact same application can have almost five times the number of context switches than it would have under optimal priority conditions.

Case	Messages	Context Switches
Case-1: Equal Priorities	9	4
Case-2: Unique Priorities	9	18

Figure 10 - When tasks are assigned unique priorities, 350% more context switches occurred, resulting in a significantly less efficient system.

In Case-2, nine messages are sent, and nine are retrieved, and eighteen context switches occur.

Timing Analysis

Of course, the major implication of the extra context switches introduced by the use of unique priorities is the additional RTOS overhead those context switches cause. In order to measure the extra overhead introduced by the additional context switches, we measure the clock count differences between the cycle-boundary relinquish events. Each event has a unique time-stamp taken from the system clock. Subtracting one "RO" time stamp from the next yields the elapsed time for the cycle. In Case-1, we have 1024 tics, at a rate of 200MHz, or 5.12µs. In Case-2, we have 1861 tics, for 9.30µs. As a result of the additional context switches and preemptions caused by the use of unique priorities in Case-2, the application suffers increased overhead.

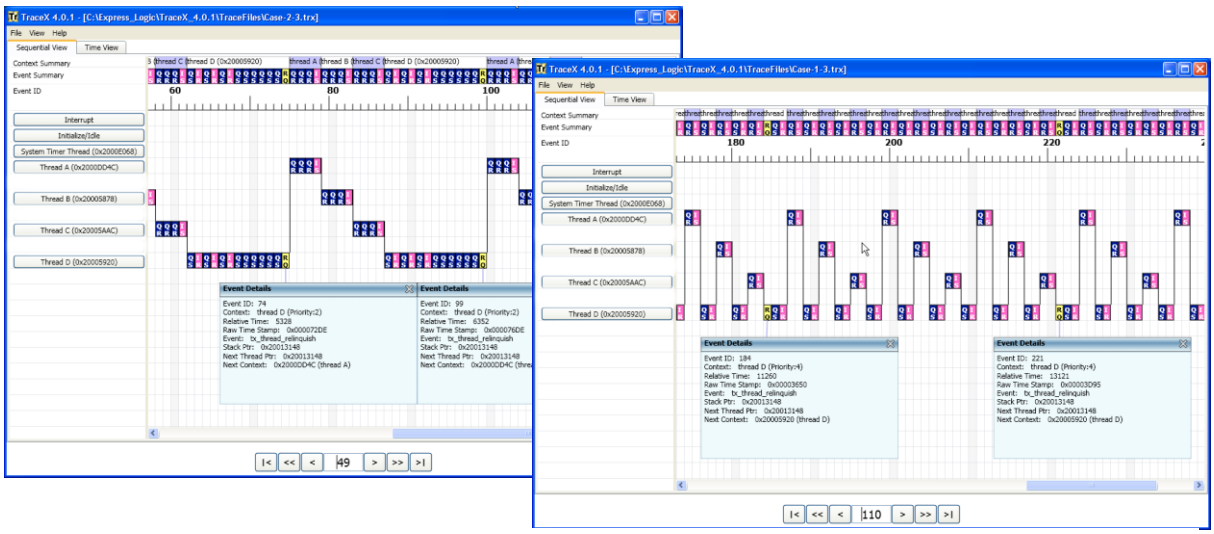

Figure 11 – Cycle Start and Cycle End event time stamps show elapsed time for a complete cycle. Case-1 (Equal Priorities) is shown on the left, while Case-2 (Unique Priorities) is shown on the right.

In this timed experiment, we used the ThreadX RTOS with a very fast 0.35µs context switch. The additional context switch operations resulting from the use of unique priorities with other RTOSes actually can add up to 250µs per context switch, depending on the RTOS used. The following results were observed:

Priority Assignment	Total Messages Per Cycle	Total Time Per Cycle	Average Time Per Message	Messages Per Millisecond (Throughput)
Case-1: Equal	9	5.074µs	0.563µs	1,776
Case-2: Unique	9	8.788µs	0.976µs	1,024

Figure 12 - Timing measurements show a significant increase in overhead with unique priorities

Observations
This experiment shows that fourteen additional context switches were performed in Case-2, and total processing time increased by more than 80%, while the exact same number of messages were sent and received. If context switch operations were not as fast as the 0.35µs in this example, the impact on total processing time would be even greater. The results here show more than an 80% increase in RTOS overhead, and a corresponding reduction in system throughput, as a result of using unique priorities for each application thread.

The use of unique priorities might also make system performance unpredictable. Loss of predictability occurs because the context switch overhead varies as a result of the sequence of thread activation, rather than in a prescribed fashion, as with the round-robin scheduling used with threads of equal priority.

The developer can eliminate this aspect of unpredictability by assigning multiple threads the same priority since there will always be a consistent number of context switches to perform a given amount of work. If distinct priorities are assigned to each thread, then the application developer needs to be acutely aware of the potential for variance in system performance and responsiveness.

When to Use Unique Priorities
While use of unique priorities might result in more context switches and reduced throughput than running multiple threads at the same priority, in some instances it is the appropriate thing to do. For example, if latency is more important than throughput, in the previous example, we would want Thread A to run as soon as a message arrives in its queue, rather than waiting for its round-robin turn. To make sure that happened, we'd make Thread A higher in priority than Thread D. Likewise with Threads B and C. We would achieve lower latency, but at the expense of cutting our throughput from 1,776 messages per millisecond to 1,024, as can be seen from the preceding table.

Priority Inheritance and Time-Slicing
Developers can best deal with the somewhat uncertain context switch overhead caused by thread priority selection by keeping as many threads as possible at the same priority level. In other words, only use different priority levels when latency outweighs throughput and preemption is absolutely required – never in any other case.

Furthermore, running multiple threads at the same priority makes it possible to properly address other system requirements such as priority inheritance, round-robin scheduling, and time-slicing. Each of these mechanisms is important in a real-time system, and each can be used to keep system overhead low and—perhaps more importantly—to keep system behavior understandable.

- **Priority Inheritance** is a mechanism employed by an RTOS to prevent deadlock in a case of priority inversion. Priority inversion occurs when a low priority thread owns a resource needed by a higher priority thread, but the lower priority thread gets preempted by a thread with a priority between the two. Thus, the low priority thread cannot run, since it has been preempted by a higher priority thread, and the resource it holds cannot be released. The high priority thread ends up waiting for the low priority thread to release the resource, effectively deadlocking the high priority thread.

Priority inheritance allows the low-priority thread to temporarily assume the priority of the high-priority thread waiting for the resource. This allows the low-priority thread to run to the point where it can release the resource, and then the high-priority thread can get it and run. If all threads had to have a unique priority, the low-priority thread could not assume the priority of the waiting thread, since that priority is already in use. Likewise, with a limit on the number of threads at a given priority, it will be impossible for the low priority thread to be raised to the level of the high priority thread if there are already the maximum number of threads at that priority. If the limit of threads at a priority is 1, then priority inheritance is never possible, and some other solution to priority inversion must be found.

- **Round-Robin Scheduling** is a method used by an RTOS to run threads in a circular fashion, letting each thread run until it becomes "blocked," or relinquishes its turn. In other words, none of the threads is preempted by a higher-priority thread. Round-robin scheduling allows multiple equally important activities to run at the same priority, while still retaining individual encapsulation. This approach is used in our Case-1 above, where all four threads have a priority of 4, and run in sequence without preemption from higher-priority threads.

- **Time Slicing** is an RTOS scheduling method that distributes CPU cycles to multiple threads in a weighted manner. Most commonly, it is used to give threads at the same priority level a certain number of CPU cycles, rotating from thread to thread and then back to the beginning of the group continuously as long as those threads remain active. It also can be used to allocate percentages of CPU time to each thread. For example, giving Thread A 25%, Thread B 10%, Thread C 10%, and Thread D 55% of the CPU cycles while that priority is active. This is generally achieved by dividing an arbitrary number of CPU cycles (a "block") into proportional parts. In this example, a block of 1000 cycles might be used (e.g., 5μs on a 200MHz system) where Thread A executes for 250 cycles, Thread B for 100 cycles, Thread C for 100 cycles, and Thread D for 550 cycles, then returning to Thread A for another 250 cycles and so on. This allocation enables system designers to provide more time for threads they determine to require more operations, but not to preempt the other threads at the same priority.

Conclusion

Assigning multiple threads the same priority can have many beneficial effects and can help system designers avoid traps that threaten the proper operation of their real-time system. In particular, it can reduce overhead, increase throughput, and enable priority inheritance and time-slicing scheduling methodologies. The developer is encouraged to use as few distinct priorities as possible and to reserve unique priorities for those instances where true preemption is required. Many RTOSes offer both priority assignment options, but some limit the number of threads at a given priority. Worse yet, some RTOSes only allow a single thread at a given priority, making it impossible for those RTOSes to support the higher-throughput round-robin scheduling approach, or to properly implement priority inheritance. It is important that the developer understand the restrictions of RTOSes that do not enable the assignment of multiple threads to the same priority, and to make the RTOS selection with this in mind.